



# 23rd International Systems and Software Product Line Conference

September 9 - 13, 2019

Paris, France



Taken from Pixabay under CC0 1.0

## Proceedings - Volume A

### EDITED BY:

Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, Tewfik Ziadi







# 23rd International Systems and Software Product Line Conference

Proceedings - Volume A

---

Gold Sponsors



---

Supporters



Centre  
de Recherche  
en Informatique

---



The Association for Computing Machinery  
1601 Broadway, 10th Floor  
New York, New York 10019, USA

**ACM COPYRIGHT NOTICE. Copyright © 2020 by the Association for Computing Machinery, Inc.** Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

For other copying of articles that carry a code at the bottom of the first or last page, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, +1-978-750-8400, +1-978-750-4470 (fax).

**ACM ISBN:** 978-1-4503-7138-4

# Table of Contents

<b>Welcome Message</b> . . . . .	<b>xi</b>
<b>Organizing Committee</b> . . . . .	<b>xii</b>
<b>Program Committees</b> . . . . .	<b>xiv</b>
<b>Keynotes</b>	
Performance Analysis for Highly-Configurable Systems . . . . .	xviii
<i>Christian Kästner</i>	
Becoming and Being a Researcher – What I Wish Someone Would Have Told me When I Started Doing Research . . . . .	xix
<i>Carlo Ghezzi</i>	
Variability Variations in Cyber-Physical Systems . . . . .	xx
<i>Lidia Fuentes</i>	
DSLs, Formal Methods, and Feature Models . . . . .	xxi
<i>Björn Engelmann</i>	
SAT Oracles, for NP-Complete Problems and Beyond . . . . .	xxii
<i>Daniel Le Berre</i>	
<b>Joint SPLC/ECSA Panel "Women in Software Engineering"</b> . . . . .	<b>xxiii</b>
<b>Testing</b>	
[Research] Automating Test Reuse for Highly Configurable Software: An Experiment . .	1
<i>Stefan Fischer, Rudolf Ramler, Lukas Linsbauer, and Alexander Egyed</i>	
[Journal First] Mutation Operators for Feature-Oriented Software Product Lines . . .	12
<i>Jacob Krüger, Mustafa Al-Hajjaji, Thomas Leich, and Gunter Saake</i>	
[Journal First] Extended Abstract of "Spectrum-Based Fault Localization in Software Prod- uct Lines" . . . . .	13
<i>Aitor Arrieta, Sergio Segura, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria</i>	

[Industry] Applying Product Line Testing for the Electric Drive System . . . . .	14
<i>Rolf Ebert, Jahir Julianis, Stefan Kriebel, Matthias Markthaler, Benjamin Pruenster, Bernhard Rumpe, and Karin Samira Salman</i>	

## Domain Implementation

[Journal First] Feature-Oriented Contract Composition . . . . .	25
<i>Thomas Thüm, Alexander Knüppel, Stefan Krüger, Stefanie Bolle, and Ina Schaefer</i>	
[Industry] Extraction of Configurable and Reusable Microservices from Legacy Systems: An Exploratory Study . . . . .	26
<i>Luiz Carvalho, Alessandro Garcia, Wesley K. G. Assunção, Rodrigo Bonifácio, Leonardo P. Tizzei, and Thelma Elita Colanzi</i>	
[Journal First] Journal First Presentation of a Comparative Study of Workflow Customiza- tion Strategies: Quality Implications for Multi-Tenant SaaS . . . . .	32
<i>Majid Makki, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen</i>	
[Industry] App Variants and Their Impact on Mobile Architecture: An Experience Report	33
<i>Marc Dahlem, Ricarda Rahm, and Martin Becker</i>	

## Solution-Space Analysis

[Research] Static Analysis of Featured Transition Systems . . . . .	39
<i>Maurice H. ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini</i>	
[Research] Learning from Difference: An Automated Approach for Learning Family Mod- els from Software Product Lines . . . . .	52
<i>Carlos Diego N. Damasceno, Mohammad Reza Mousavi, and Adenilso Simao</i>	
[Journal First] Feature-Family-Based Reliability Analysis of Software Product Lines . . .	64
<i>Andre Lanna, Thiago Castro, Vander Alves, Genaina Rodrigues, Pierre-Yves Schobbens, and Sven Apel</i>	
[Research] Variability-Aware Semantic Slicing Using Code Property Graphs . . . . .	65
<i>Lea Gerling and Klaus Schmid</i>	

## Challenges and Solutions

[Challenge Proposal] Applying Product Line Engineering Concepts to Deep Neural Net- works . . . . .	72
<i>Javad Ghofrani, Ehsan Kozegar, Anna Lena Fehlhaber, and Mohammad Divband Soorati</i>	

[Challenge Proposal] Product Sampling for Product Lines: The Scalability Challenge . . . . .	78
<i>Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer</i>	
[Challenge Solution] t-wise Coverage by Uniform Sampling . . . . .	84
<i>Jeho Oh, Paul Gazzillo, and Don Batory</i>	
[Challenge Solution] A Graph-Based Feature Location Approach Using Set Theory . . . . .	88
<i>Richard Müller and Ulrich Eisenecker</i>	
[Challenge Solution] Comparison-Based Feature Location in ArgoUML Variants . . . . .	93
<i>Gabriela Karoline Michelon, Lukas Linsbauer, Wesley K. G. Assunção, and Alexander Egyed</i>	
[Challenge Solution] Migrating Java-Based Apo-Games into a Composition-Based Software Product Line . . . . .	98
<i>Jamel Debbiche, Oskar Lignell, Jacob Krüger, and Thorsten Berger</i>	
[Challenge Solution] Migrating the Android Apo-Games into an Annotation-Based Software Product Line . . . . .	103
<i>Jonas Åkesson, Sebastian Nilsson, Jacob Krüger, and Thorsten Berger</i>	

## **Emerging Application Areas**

[Research] DNA as Features: Organic Software Product Lines . . . . .	108
<i>Mikaela Cashman, Justin Firestone, Myra B. Cohen, Thammasak Thianniwit, and Wei Niu</i>	
[Research] Automated Search for Configurations of Convolutional Neural Network Architectures . . . . .	119
<i>Salah Ghamizi, Maxime Cordy, Mike Papadakis, and Yves Le Traon</i>	
[Research] Piggyback IDE Support for Language Product Lines . . . . .	131
<i>Thomas Kühn, Walter Cazzola, Nicola Pirritano Giampietro, and Massimiliano Poggi</i>	
[Industry] Industrial Perspective on Reuse of Safety Artifacts in Software Product Lines .	143
<i>Christian Wolschke, Martin Becker, Sören Schneickert, Rasmus Adler, and John MacGregor</i>	

## **Community Efforts**

[Industry] How Domain-Specific Modeling Languages Address Variability in Product Line Development: Investigation of 23 Cases . . . . .	155
<i>Juha-Pekka Tolvanen and Steven Kelly</i>	
[Research] Software Product Line Engineering: A Practical Experience . . . . .	164
<i>Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes</i>	

[Research] Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems . . . . .	177
<i>Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Javier Martinez, and Thorsten Berger</i>	
[Research] Industrial and Academic Software Product Line Research at SPLC: Perceptions of the Community . . . . .	189
<i>Rick Rabiser, Klaus Schmid, Martin Becker, Goetz Botterweck, Matthias Galster, Iris Groher, and Danny Weyns</i>	

## Requirements Engineering

[Industry] Feature Oriented Refinement from Requirements to System Decomposition: Quantitative and Accountable Approach . . . . .	195
<i>Masaki Asano, Yoichi Nishiura, Tsuneo Nakanishi, and Keiichi Fujiwara</i>	
[Journal First] Enabling Automated Requirements Reuse and Configuration . . . . .	206
<i>Yan Li, Tao Yue, Shaukat Ali, and Li Zhang</i>	

## Metrics and Refactoring

[Research] Formal Foundations for Analyzing and Refactoring Delta-Oriented Model-Based Software Product Lines . . . . .	207
<i>Christopher Pietsch, Udo Kelter, Timo Kehrer, and Christoph Seidl</i>	
[Research] Covert and Phantom Features in Annotations: Do They Impact Variability Analysis? . . . . .	218
<i>Kai Ludwig, Jacob Krüger, and Thomas Leich</i>	
[Research] Identifying and Visualizing Variability in Object-Oriented Variability-Rich Systems . . . . .	231
<i>Xhevahire Térnava, Johann Mortara, and Philippe Collet</i>	
[Journal First] Metrics for Analyzing Variability and Its Implementation in Software Product Lines: A Systematic Literature Review . . . . .	244
<i>Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid</i>	

## Feature-Model Evolution

[Research] Semantic Evolution Analysis of Feature Models . . . . .	245
<i>Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpf</i>	

[Journal First] Achieving Change Requirements of Feature Models by an Evolutionary Approach . . . . .	256
<i>Paolo Arcaini, Angelo Gargantini, and Marco Radavelli</i>	

[Research] Foundations of Collaborative, Real-Time Feature Modeling . . . . .	257
<i>Elias Kuiter, Sebastian Krieter, Jacob Krüger, Thomas Leich, and Gunter Saake</i>	

## Configuration and Sampling

[Research] Process Mining to Unleash Variability Management: Discovering Configuration Workflows Using Logs . . . . .	265
<i>Ángel Jesús Varela-Vaca, José A. Galindo, Belén Ramos-Gutiérrez, María Teresa Gómez-López, and David Benavides</i>	

[Research] Towards Quality Assurance of Software Product Lines with Adversarial Configurations . . . . .	277
<i>Paul Temple, Mathieu Acher, Gilles Perrouin, Battista Biggio, Jean-Marc Jézéquel, and Fabio Roli</i>	

[Research] Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features . . . . .	289
<i>Daniel-Jesús Muñoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory</i>	

## Problem-Space Analysis

[Journal First] Automated Analysis of Feature Models: Quo Vadis? . . . . .	302
<i>José A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés</i>	

[Research] A Kconfig Translation to Logic with One-Way Validation System . . . . .	303
<i>David Fernandez-Amoros, Ruben Heradio, Christoph Mayr-Dorn, and Alexander Egyed</i>	

[Industry] Using Relation Graphs for Improved Understanding of Feature Models in Software Product Lines . . . . .	309
<i>Slawomir Duszynski, Saura Jyoti Dhar, and Tobias Beichter</i>	

## Workshops

Second International Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution 2019) . . . . .	320
<i>Michael Nieke, Lukas Linsbauer, Jacob Krüger, and Thomas Leich</i>	

Second International Workshop on Experiences and Empirical Studies on Software Reuse (WEESR 2019) . . . . .	321
<i>Jaime Chavarriaga and Julio Ariel Hurtado</i>	
Fourth International Workshop on Software Product Line Teaching (SPLTea 2019) . . . . .	322
<i>Mathieu Acher, Rick Rabiser, and Roberto E. Lopez-Herrejon</i>	
First International Workshop on Languages for Modelling Variability (MODEVAR 2019) . .	323
<i>David Benavides, Rick Rabiser, Don Batory, and Mathieu Acher</i>	
Seventh International Workshop on Reverse Variability Engineering (REVE 2019) . . . . .	324
<i>Mathieu Acher, Tewfik Ziadi, Roberto E. Lopez-Herrejon, and Jaber Martinez</i>	

## Tutorials

Machine Learning and Configurable Systems: A Gentle Introduction . . . . .	325
<i>Hugo Martin, Juliana Alves Pereira, Mathieu Acher, and Paul Temple</i>	
Software Reuse for Mass Customization . . . . .	327
<i>Mike Mannion and Hermann Kaindl</i>	
Variability Modeling and Implementation with EASy-Producer . . . . .	328
<i>Klaus Schmid, Holger Eichelberger, and Sascha El-Sharkawy</i>	
Describing Variability with Domain-Specific Languages and Models . . . . .	329
<i>Juha-Pekka Tolvanen and Steven Kelly</i>	
Automated Evaluation of Embedded-System Design Alternatives . . . . .	330
<i>Maxime Cordy and Sami Lazreg</i>	
Feature-Based Systems and Software Product Line Engineering: PLE for the Enterprise .	331
<i>Charles W. Krueger and Paul C. Clements</i>	
Variability Modeling and Management of MATLAB/Simulink Models . . . . .	332
<i>Aitor Arrieta</i>	

## Welcome Message

Welcome to SPLC 2019, the 23rd International Systems and Software Product Line Conference. Product lines represent one of the most exciting paradigm shift in software and systems development, with new challenges and opportunities for both research and practice. For 23 years, SPLC has been the meeting ground for practitioners, researchers, and educators interested in systems and software product lines. SPLC is a great venue for learning about the state of the art as well as practice, trends, innovations, industry experiences, and challenges in the area of systems family engineering at large. SPLC 2019 took place from September 9th to 13th, 2019 in the vibrant city of Paris.

For the first time, we co-located SPLC with ECSA, the 13th European Conference on Software Architecture. We gave the opportunity to participants of both conferences to meet in Paris around a set of common events including two joint keynotes and a common doctoral symposium. Furthermore, we organized a joint panel on Women in Software Engineering, which provided a forum for discussions on how to achieve more diversity in software engineering.

For participants, SPLC 2019 proposed a very exciting program of top notch research and industry papers as well as journal first presentations, workshops, tutorials, challenges, demonstrations, doctoral proposals, artifacts, and great keynote presentations. SPLC received 115 submissions: 45 research papers and 13 research artifacts, 10 industry papers, 17 journal-first presentations, 7 workshop proposals, 2 challenge proposals and 6 solutions, 13 demo and tool papers, 3 doctoral proposals and 7 tutorials. Based on at least three reviews and intensive discussions, the committees selected 15 full-paper contributions for the research track and 5 for the industry track, leaving us with acceptance rates of 33% and 50%, respectively. We are especially grateful to all members of the program committees for helping us to seek submissions and provide valuable and constructive reviews.

We would like to thank our keynote speakers Christian Kästner, Lidia Fuentes, Björn Engelmann, Daniel Le Berre, Carlo Ghezzi, who graciously agreed to share their perspectives, experiences, and insights with the community. The program committee members and track chairs deserve a mention for their hard work in reviewing and discussing the papers. Our thanks also for the local organizing team whose efforts were instrumental for ensuring the success of the conference. Finally, we would like to thank our sponsors and institutional partners for their support and contributions. These include Laboratoire d’Informatique de Paris 6 (LIP6), Sorbonne University, Centre de Recherche en Informatique (CRI), University of Paris 1 Pantheon Sorbonne, pure-systems GmbH, and BigLever Software Inc.

Sincerely,

Laurence Duchien, Thomas Thüm, Camille Salinesi, Tewfik Ziadi, Patrick Heymans, Thomas Fogdal, Jabier Martinez, Timo Kehrer, Leopoldo Teixeira, Rick Rabiser, Carlos Cetina, Christoph Seidl, Raúl Mazo, Philippe Collet, Leticia Montalvillo, Oscar Díaz, Marianne Huchard, Thorsten Berger, Natsuko Noda, Goetz Botterweck, Lom Messan Hillah, Jacques Robin, Clément Quinton, Wesley K. G. Assunção, Anas Shatnawi, Xhevahire Térnavá, Juliana Alves Pereira, Thùy Dodo

# **Organizing Committee**

## **General Chairs**

Camille Salinesi, *University Paris 1 Panthéon-Sorbonne, France*  
Tewfik Ziadi, *Sorbonne University, France*

## **Research Track Chairs**

Laurence Duchien, *University Lille, France*  
Thomas Thüm, *TU Braunschweig, Germany*

## **Industrial Systems and Software Product Lines Chairs**

Patrick Heymans, *University of Namur, Belgium*  
Thomas Fogdal, *Danfoss Power Electronics A/S, Denmark*

## **Challenge Track Chairs**

Jabier Martinez, *Tecnalia, Spain*  
Timo Kehrer, *Humboldt-Universität zu Berlin, Germany*

## **Demonstrations and Tools Chairs**

Leopoldo Teixeira, *Federal University of Pernambuco, Brazil*  
Rick Rabiser, *Johannes Kepler University Linz, Austria*

## **Workshops Chairs**

Carlos Cetina, *University San Jorge, Spain*  
Christoph Seidl, *TU Braunschweig, Germany*

## **Journal First Chair**

Raúl Mazo, *University Paris 1 Panthéon-Sorbonne, France*

## **Tutorials Chairs**

Philippe Collet, *Université Côte d'Azur, France*  
Leticia Montalvillo, *IK4-IKERLAN Research Center, Spain*

## **Doctoral Symposium Chairs**

Oscar Díaz, *University of the Basque Country, Spain*  
Marianne Huchard, *University of Montpellier, France*

## **Panels Chair**

Thorsten Berger, *Chalmers / University of Gothenburg, Sweden*

## **Hall of Fame Chairs**

Natsuko Noda, *Shibaura Institute of Technology, Japan*  
Goetz Botterweck, *University of Limerick, Ireland*

## **Local Organizing Chairs**

Lom Messan Hillah, *University Paris Nanterre and Sorbonne University, France*  
Jacques Robin, *University Paris 1 Panthéon-Sorbonne, France*

## **Publicity and Social Media Chairs**

Clément Quinton, *University Lille, France*  
Wesley K. G. Assunção, *Federal University of Technology – Paraná, Brazil*

## **Web Chairs**

Anas Shatnawi, *Sorbonne University, France*  
Aleksandr Chueshev, *Sorbonne University, France*

## **Proceedings Chair**

Xhevahire Ternava, *Sorbonne University, France*

## **Student Volunteers Chair**

Juliana Alves Pereira, *University of Rennes 1, France*

## **Finance Chair**

Thùy Dodo, *Sorbonne University, France*

## Program Committees

### Research Track

Mathieu Acher	University of Rennes 1, France
Mustafa Al-Hajjaji	pure-systems GmbH, Germany
Shaukat Ali	Simula Research Laboratory, Norway
Vander Alves	University of Brasília, Brazil
Paolo Arcaini	National Institute of Informatics, Japan
Ebrahim Bagheri	Ryerson University, Canada
Maurice H. ter Beek	ISTI–CNR, Italy
David Benavides	University of Seville, Spain
Thorsten Berger	Chalmers   University of Gothenburg, Sweden
Goetz Botterweck	University of Limerick, Ireland
Rafael Capilla	Universidad Rey Juan Carlos, Spain
Rubby Casallas	Universidad de los Andes, Colombia
Walter Cazzola	Università degli Studi di Milano, Italy
Carlos Cetina	San Jorge University, Spain
Marsha Chechik	University of Toronto, Canada
Jane Cleland-Huang	University of Notre Dame, USA
Philippe Collet	Université Côte d'Azur, France
Ferruccio Damiani	University of Torino, Italy
Xavier Devroey	TU Delft, Netherlands
Oscar Díaz	University of Basque Country, Spain
Alessandro Fantechi	Università di Firenze, Italy
Alexander Felfernig	TU Graz, Austria
Lidia Fuentes	University of Málaga, Spain
Matthias Galster	University of Canterbury, New Zealand
Marianne Huchard	Université de Montpellier, France
Christian Kästner	Carnegie Mellon University, USA
Timo Kehrer	Humboldt-Universität zu Berlin, Germany
Jacques Klein	University of Luxembourg, Luxembourg
Dimitri Van Landuyt	Katholieke Universiteit Leuven, Belgium
Jaejoon Lee	Lancaster University, United Kingdom
Jihyun Lee	ChonBuk National University, South Korea
Kwanwoo Lee	Hansung University, South Korea
Thomas Leich	Harz University, Germany

Malte Lochau	TU Darmstadt, Germany
Roberto E. Lopez-Herrejon	Université du Québec, Canada
Ivan do Carmo Machado	Federal University of Bahia, Brazil
Mike Mannion	Glasgow Caledonian University, United Kingdom
Tomi Männistö	University of Helsinki, Finland
Jabier Martinez	Tecnalia, Spain
Raúl Mazo	University Paris 1 Panthéon-Sorbonne, France
Mohammad Reza Mousavi	University of Leicester, United Kingdom
Jennifer Perez	Universidad Politecnica de Madrid, Spain
Gilles Perrouin	University of Namur, Belgium
Clément Quinton	University Lille, France
Rick Rabiser	Johannes Kepler University Linz, Austria
Iris Reinhartz-Berger	University of Haifa, Israel
Márcio Ribeiro	Federal University of Alagoas, Brazil
Klaus Schmid	University of Hildesheim, Germany
Sandro Schulze	University of Magdeburg, Germany
Christoph Seidl	TU Braunschweig, Germany
Daniel Strüber	Chalmers   University of Gothenburg, Sweden
Leopoldo Teixeira	Federal University of Pernambuco, Brazil
Andrea Vandin	DTU Compute, Denmark
Élise Vareilles	IMT Mines Albi, France
Eric Walkingshaw	Oregon State University, USA
Yue Wang	Hang Seng Management College, Hong Kong
Tao Yue	University of Oslo, Norway
Tewfik Ziadi	Sorbonne University, France

## Artifacts Evaluation

Paola Accioly	Federal University of Pernambuco, Brazil
Davide Basile	University of Florence, Italy
Jessie Carbonnel	University of Montpellier, France
Maxime Cordy	University of Luxembourg, Luxembourg
Clemens Dubslaff	University of Dresden, Germany
José A. Galindo	University of Seville, Spain
José Miguel Horcas	University of Málaga, Spain
Sebastian Krieter	Harz University, Germany
Jacob Krüger	University of Magdeburg, Germany
Lukas Linsbauer	Johannes Kepler University Linz, Austria
Jens Meinicke	Carnegie Mellon University, USA
Juan C. Muñoz-Fernández	ICESI, Colombia
Damir Nesic	KTH University, Sweden
Michael Nieke	TU Braunschweig, Germany
Juliana Alves Pereira	University of Rennes 1, France

Gabriela Sampaio  
Paul Temple  
Lina Ochoa Venegas

Imperial College London, England  
University of Namur, Belgium  
Universidad de los Andes, Colombia

## Industrial Systems and Software Product Lines Track

Vander Alves	University of Brasília, Brazil
Martin Becker	Fraunhofer IESE, Germany
Danilo Beuche	pure-systems GmbH, Germany
Jan Bosch	Chalmers University of Technology, Sweden
Goetz Botterweck	University of Limerick, Ireland
Paul Clements	BigLever Software, Inc, USA
Maxime Cordy	University of Luxembourg, Luxembourg
Deepak Dhungana	Siemens AG, Austria
Christoph Elsner	Siemens AG, Germany
Lidia Fuentes	University of Málaga, Spain
Sebastien Gerard	CEA, LIST, France
Paul Grünbacher	Johannes Kepler University Linz, Austria
Oystein Haugen	Østfold University College, Norway
Jean-Marc Jézéquel	University of Rennes 1, France
Frank Van Der Linden	Philips Healthcare, Netherlands
Steve Livengood	HP, Inc, USA
Tomi Männistö	University of Helsinki, Finland
Andreas Rummel	SAP AG, Germany
Daniel Schall	Siemens Corporate Technology, Germany
Bran Selic	Malina Software Corp., Canada
Hailong Sun	Beihang University, China
Juha-Pekka Tolvanen	MetaCase, Finland
Hironori Washizaki	Waseda University, Japan

## Challenge Proposals and Solutions Track

Wesley K. G. Assunção  
Jaime Font  
Jacob Krüger  
Lukas Linsbauer  
Malte Lochau  
Roberto E. Lopez-Herrejon  
Tobias Pett  
Anas Shatnawi  
Thomas Thüm

Federal University of Technology - Paraná, Brazil  
University of San Jorge, Spain  
University of Magdeburg, Germany  
Johannes Kepler University Linz, Austria  
TU Darmstadt, Germany  
Université du Québec, Canada  
TU Braunschweig, Germany  
Sorbonne University, France  
TU Braunschweig, Germany

## **Journal First Paper Track**

Eduardo Almeida	Federal University of Bahia, Brazil
Aitor Arrieta	Mondragon University, Spain
David Benavides	University of Seville, Spain
Thorsten Berger	Chalmers   University of Gothenburg, Sweden
Rafael Capilla	Universidad Rey Juan Carlos, Spain
Iris Groher	Johannes Kepler University Linz, Austria
Jean-Marc Jezequel	University of Rennes 1, France
Nicole Levy	Cedric, CNAM, France
Roberto E. Lopez-Herrejon	Université du Québec, Canada
Klaus Schmid	University of Hildesheim, Germany
Abdelhak-Djamel Seriai	University of Montpellier, France

## **Subreviewers**

Hugo Araujo	Universidade Federal de Pernambuco, Brazil
Carlos Diego Damasceno	University of Leicester, United Kingdom
Swaib Dragule	Chalmers   University of Gothenburg, Sweden
Sascha El-Sharkawy	University of Hildesheim, Germany
Stefania Gnesi	ISTI-CNR, Italy
Ruben Heradio	Universidad Nacional de Educacion a Distancia Madrid, Spain
Emilio Incerto	IMT Lucca, Italy
Pooyan Jamshidi	University of South Carolina, USA
Alexander Knüppel	TU Braunschweig, Germany
Sebastian Krieter	Harz University, Germany
Jacob Krüger	University of Magdeburg, Germany
Michael Lienhardt	University of Turin, Italy
Hong Lu	Simula Research Laboratory, Norway
Kai Ludwig	Harz University, Germany
Lars Luthmann	TU Darmstadt, Germany
Wardah Mahmood	Chalmers   University of Gothenburg, Sweden
Jacopo Mauro	University of Southern Denmark, Denmark
Mukelabai Mukelabai	Chalmers   University of Gothenburg, Sweden
Tobias Pett	TU Braunschweig, Germany
Dennis Reuling	Universität - GH - Siegen, Germany
Sebastian Ruland	TU Darmstadt, Germany
Shmuel Tyszberowicz	University of Torino, Italy
Mahsa Varshosaz	Halmstad University, Sweden
Markus Weckesser	TU Darmstadt, Germany

# Performance Analysis for Highly-Configurable Systems

Christian Kästner, *Carnegie Mellon University, Pittsburgh, USA*

**Abstract.** Almost every modern software system is highly configurable with dozens or more options to customize behavior for different use cases. Beyond enabling or disabling optional functionality, configuration options often adjust tradeoffs among accuracy, performance, security, and other qualities. However with possible interactions among options and an exponentially exploding configuration space, reasoning about the impact of configurations is challenging. Which options affect performance or accuracy? Which options interact? What's the optimal configuration for a given workload? In this talk, I will give an overview of different strategies and challenges to learn performance models from highly-configurable systems by observing their behavior in different configurations, looking at sampling and learning strategies, transfer learning strategies, and strategies that analyze the internals or architecture of the system.

**About Christian Kästner.** Christian Kästner is an associate professor in the School of Computer Science at Carnegie Mellon University. He received his PhD in 2010 from the University of Magdeburg, Germany, for his work on virtual separation of concerns. For his dissertation he received the prestigious GI Dissertation Award. Kästner develops mechanisms, languages, and tools to implement variability in a disciplined way despite imperfect modularity, to understand feature interactions and interoperability issues, to detect errors, to help with nonmodular changes, and to improve program comprehension in software systems, typically systems with a high amount of variability. Among others, Kästner has developed approaches to parse and type check all compile-time configurations of the Linux kernel in the TypeChef project.



# Becoming and Being a Researcher – What I Wish Someone Would Have Told me When I Started Doing Research

**Carlo Ghezzi, Politecnico di Milano, Italy**

**Abstract.** Why should one wish to become a researcher? What is the role of research and researchers in society? What does one need to do to become a researcher as a PhD student (but also before and after)? What can be the progress of a researcher in his or her career? How to survive and be successful? These are some of the questions I will try to answer in my presentation, based on what I learnt from others and on my own experience. Very often, young researchers are too busy doing their own research and don't care about the global picture and ignore these questions. Often, their academic supervisors only focus on the technical side of their supervision, and don't open the eyes of their young research collaborators. But then, sooner or later, these questions emerge and an answer must be given. In particular, I will focus on three issues:

1. Diffusion of research, through publications and other means. What does a beginning researcher need to know and what is a good personal strategy?
2. Evaluation of research and researcher. Researchers need to understand that they will be subject to continuous evaluation. Why? How? And, most important, how should they prepare to live through continuous evaluations?
3. Ethics. Researchers need to be aware of the ethical issues involved in doing research. On the one side, integrity is needed in the every-day practice of research. On the other, research is changing the world in which we live. The products of research lead to innovations that can have profound influence on society, and because of the increasingly fast transition from research to practice, they affect the world even before we understand the potential risks. What researchers might see as purely technical problems may have ethical implications, and this requires ethics awareness while doing research.

**About Carlo Ghezzi.** Carlo Ghezzi is an ACM Fellow (1999), an IEEE Fellow (2005), a member of the European Academy of Sciences and of the Italian Academy of Sciences. He received the ACM SIGSOFT Outstanding Research Award (2015), the Distinguished Service Award (2006), and the 2018 TCSE Distinguished Education Award from IEEE Computer Society Technical Council on Software Engineering (TCSE). He has been President of Informatics Europe. He has been a member of the program committee of flagship conferences in the software engineering field, such as the ICSE and ESEC/FSE, for which he also served as Program and General Chair. He has done research in programming languages and software engineering for over 40 years and has been a recipient of an ERC Advanced Grant on self-adaptive software systems. He has published over 200 papers in international journals and conferences and co-authored 6 books.



# Variability Variations in Cyber-Physical Systems

**Lidia Fuentes, CAOSD group, University of Málaga, Spain**

**Abstract.** With the increasing size and heterogeneity of systems (e.g., IoT, Cyber-Physical Systems) and enhanced power and versatility of IoT devices (e.g., smart watches, home intelligence sensors), the complexity of managing different kinds of variability for a given vertical domain becomes more difficult to handle. The structural variability of cyber-physical systems becomes more complex, comprising not only the inherent hardware variability of IoT devices and their network access protocols, but also the infrastructure variability derived from modern virtualization technologies. Variability of software frameworks used to develop domain specific applications and/or services for Cloud/Edge computing environments should not be intermingled with hardware, and infrastructure variability modelling. In addition, to exploit the full potential of flexibility in processing, data storage and networking resource management, experts should define dynamic configuration processes that optimise QoS such as energy efficiency or latency respecting application-specific requirements. In this keynote talk, I will present how QoS assurance in cyber-physical systems implies modelling and configuring different kinds of variability during design, but also at runtime (e.g., user demands, usage context variability), enabling the late binding of dynamic variation points, distributed in IoT/Edge/Cloud devices.

**About Lidia Fuentes.** Lidia Fuentes is a professor at the School of Informatics at the University of Málaga, Spain since 2011, with more than twenty-five years of experience teaching, leading research projects and supervising thesis. She leads a cross-disciplinary research group CAOSD, focused on applying advanced Software Engineering Technologies to Network and Distributed systems. Her current research interests include modelling different kinds of variability of Internet of Things (IoT), and Cypher physical systems to support dynamic reconfiguration and green computing. Her scientific production has been very prolific so far, with more than two hundred scientific publications in international forums. Her work has received several best-paper awards at conferences such as ICSR or SPLC-Tools track. She chaired several conferences as general chair (Modularity 2016), program chair (SPLC industry track, VaMoS, ...), served on numerous program committees, and also participated as a panellist at ICSR 2017 and is member of the steering committee of VaMoS.



# DSLs, Formal Methods, and Feature Models

Björn Engelmann, *Germany*

**Abstract.** At the core of Model-driven development, there is the idea of generalizing software development processes and tools to arbitrary domains. To this end, Domain-Specific Languages (DSLs) provide the intellectual tooling necessary for said domains and language workbenches like JetBrains MPS allow lifting the tooling invented and developed for programming languages to these DSLs. Formal Methods represent the most powerful tooling ever developed for tackling complex problems. Originally intended for Software Engineers, these tools today find a wide range of applications across different domains ranging from mathematics to biochemistry. Integrating Formal Methods into DSLs contributes to this development and makes the tooling for tackling complex problems available to practitioners in domains other than Software Engineering. Feature Models and their Configurations provide the intellectual tooling for the domain of Variability. At my last employer, me and my team developed a Variability DSL based on MPS thus allowing the application of the advanced tool support known from Software Development to Feature Modelling. In particular, it becomes possible to automatically ensure various properties of the models and configurations using the SMT solver Z3. In this talk I would like to share some insights gained from the development of said tooling as well as from its application to real-world feature models.

**About Björn Engelmann.** Björn Engelmann holds a PhD in Formal Methods from the University of Oldenburg, Germany. During his work as a language engineer, he was responsible for integrating Formal Methods into Domain Specific Languages based on MPS. He developed an Integration for SMT Solvers into MPS and supported other language engineering teams in making their DSLs solver-enabled. Amongst others, he developed the solver-enabled analyses that are part of the Variability DSL and maintains them since.



# SAT Oracles, for NP-Complete Problems and Beyond

Daniel Le Berre, *Université d'Artois et CNRS, France*

**Abstract.** SAT solvers have been used in many areas, including software product lines [1], as generic engine to solve NP-complete problems since the 2000's [2]. While the raw performances of the solvers to tackle NP-complete problems have been increasing steadily in the past [4], most impressive recent results rely on sophisticated encodings and better interaction with the solver, allowing to solve NP-hard – and even PSPACE-complete – problems. This is the case for core-based MAXSAT solvers for instance [6], and more recently SAT-based QBF solvers [3]. In this talk, I will review the current features found in modern SAT solvers, and how they are used to solve NP-hard problems. I will present a generic approach called RECAR (Recursive Explore and Check Abstraction Refinement) [5] which allows to take advantage of the feedback of the SAT solver to drive the search on a subproblem to prevent when possible memory blowup.

**About Daniel Le Berre.** Daniel Le Berre is professor of computer science at The University of Artois, France. He is interested in the practical aspects of Boolean reasoning, namely SAT, pseudo-Boolean and MAXSAT solvers. He created and maintains Sat4j, an open source java library dedicated to the resolution of Boolean satisfaction and optimization problems. He co-organized the SAT competition from 2002 to 2011 and the Pragmatics of SAT series of workshops since 2010. He is currently the editor-in-chief of the Journal on Satisfiability, Boolean Modeling and Computation (JSAT).



## References

- [1] Don S. Batory, *Feature models, grammars, and propositional formulas*, Proceedings of SPLC 2005, 2005, pp. 7–20.
- [2] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (eds.), *Handbook of satisfiability*, Frontiers in Artificial Intelligence and Applications, vol. 185, IOS Press, 2009.
- [3] Mikolás Janota, William Klieber, João Marques-Silva, and Edmund M. Clarke, *Solving QBF with counterexample guided refinement*, Artif. Intell. **234** (2016), 1–25.
- [4] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon, *The international SAT solver competitions*, AI Magazine **33** (2012), no. 1.
- [5] Jean-Marie Lagniez, Daniel Le Berre, Tiago de Lima, and Valentin Montmirail, *A recursive shortcut for CEGAR: application to the modal logic K satisfiability problem*, Proceedings of IJCAI 2017, 2017, pp. 674–680.
- [6] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva, *Iterative and core-guided maxsat solving: A survey and assessment*, Constraints **18** (2013), no. 4, 478–534.

## Joint SPLC/ECSA Panel “Women in Software Engineering”

The panel “Women in Software Engineering” is a joint panel of SPLC and the co-located conference 13th European Conference on Software Architecture (ECSA). It is organized by the chairs of ECSA’s “Women in Software Architecture” track *Anne Koziolek* (Karlsruhe Institute of Technology, Germany) and *Patrizia Scandurra* (University of Bergamo, Italy), together with the SPLC panels chair *Thorsten Berger* (Chalmers | University of Gothenburg). The panel lasts one hour, in which the panelists introduce themselves, discuss a range of problems (e.g., gender stereotypes and sexism in IT recruitment and workplaces) and solutions (e.g., quota, role models or gender equality plans) to support women in the field of software engineering. Since the audience of the two conferences is male-dominated, the panel also elicits common pitfalls and typical (perhaps unwanted) negative behavior of men together with actionable recommendations on how to avoid such. A planned outcome is a feature model representing the discussed problems and solutions.

We are delighted that the following five high-profile panelists accepted our invitation:

**Isabelle Collet**, Professor at University of Genève, Switzerland, is a former computer scientist. She received her Ph.D. in education from the University of Paris Nanterre, France, in 2005, and the following year published the book *Does Computer Science Have a Gender? Hackers, Myths and Realities*, based on her thesis work, which received an award from the French Academy of Moral and Political Sciences. She founded ARGEF, the Research Association for Gender in Education and Training, as well as a peer-reviewed, French-language science journal, *La Revue Genre Éducation Formation* (The Gender Education and Training Journal). Since 2000, she has been involved in several Europe-based projects on gender and information technology, and is an evaluator for the European Union’s GENDER-NET Plus program. She has served as a scientific expert on numerous gender-neutral education projects in STEM in Belgium, France, and Switzerland. Her research interests are focused on closing the STEM gender gap and developing inclusion strategies for women in higher education. She is also a consultant on gender-inclusive teaching for CERN in Geneva, Switzerland.



**Chiara Condi**, activist for women’s empowerment and founder of *Led By HER*, France, is an advocate and expert on gender diversity, innovation, and women’s entrepreneurship. *Led By HER* is a nonprofit that works on advocacy and programs to improve access to women’s entrepreneurship, innovation and women’s rights. In partnership with business schools and corporates, the organization delivers an entrepreneurship program to help women who have suffered from violence rebuild their lives through entrepreneurship. With a community of over 250 volunteers the organization offers courses, individual mentoring, and hackathon events co-organized with the entrepreneurial ecosystem. *Led By HER* recently launched *FoundHER-Lab*, a digital platform coded by Capgemini, to mobilize companies worldwide around women’s entrepreneurship by offering pro-bono skill based volunteering to women entrepreneurs locally. Given *Led By HER*’s impact on society, Chiara is regularly invited as a voice in the media, and the exemplary work of *Led By HER* has been featured in over 100 publications in France and abroad. Chiara is an internationally renowned speaker on gender equality, women’s entrepreneurship, the future of work, and building a more inclusive world for women. In 2017 she was awarded the Woman of Influence Prize (Business and Hope). Prior to founding *Led By HER*, Chiara worked at the European Bank for Reconstruction and Development to maximize the gender impact of the Bank’s investments, where she became inspired to launch *Led By HER* to work with women on the ground. She graduated from Harvard and has a Master’s from Sciences Po/LSE.



**Elisabeth Kohler**, Director of the CNRS Mission for Women's Integration, France, is a senior science policy officer who heads the CNRS Mission for Women's Integration since January 2018. She is in charge of designing, implementing and evaluating gender equality plans and of promoting the integration of gender and sex dimension in research. Interacting with other national stakeholder, she also develops programs to attract more women to study STEM. At European level, she coordinates the GENDER-NET Plus project aiming at integrating the gender dimension in projects relating to the UN Sustainable Development Goals. At international level, she is involved in gender equality actions with several countries (Canada, India, and Lebanon). With an academic background in cultural studies, she has previously worked in different fields at CNRS, ranging from scientific information to innovation policies, European project management and international cooperation.



**Florence Sedes**, Professor at University of Toulouse, France, works in the research area of data science. She has been active in database and information system research since her PhD in 1987. She has published over one hundred papers, books and book chapters since 2000 and advised more than 30 PhD. She heads her team in the IRIT lab (UMR CNRS 5505). She has been leading international, European and national projects on personal (meta)data privacy and management, CCTV and forensic, IoT and security, geospatial and indoor/outdoor data, and social networks, with applications via deep/machine learning for alert, spam and rumors detection, social emotion and interaction. Her research interests include modelling, developing, evaluating, and characterizing (big) data systems and techniques from both problem-driven and technique-driven perspectives. She has been heavily involved in designing data sets and platforms in order to enable assessment of the various contributions, software and systems of our community. Florence Sedes heads the french IFIP (International Federation for Information Processing) committee. She founded the "femmes&Informatique" initiative of the French Informatics Society, and the WIE French chapter of IEEE.



**Serge Abiteboul**, researcher at Inria and ENS, Paris, France, obtained a PhD from the University of Southern California and a State Doctoral Degree from the University of Paris-Sud. He has been a researcher at the Institut National de Recherche en Informatique et Automatique since 1982, a Directeur de Recherche CE since 2004, and is in a research team located at Ecole Normale Supérieure de Paris since 2016. He is Distinguished Affiliated Professor at Ecole Normale Supérieure de Cachan. He was a Lecturer at the École Polytechnique and Visiting Professor at Stanford and Oxford University. He has been Chair Professor at Collège de France in 2011–2012 and Francqui Chair Professor at Namur University in 2012–2013. He co-founded the company Xyleme in 2000. Serge Abiteboul has received the ACM SIGMOD Innovation Award in 1998, the EADS Award from the French Academy of sciences in 2007, the Milner Award from the Royal Society in 2013, and a European Research Council Fellowship (2008–2013). He became a member of the French Academy of Sciences in 2008 and a member of the Academy of Europe in 2011. He has been a member of the Conseil national du numérique (2013–2016) and chairman of the scientific board of the Société d'Informatique de France (2013–2015). He is Chair of the Strategic Council of the Blaise Pascal Foundation. His research work focuses mainly on data, information and knowledge management, particularly on the web. Serge Abiteboul founded, and is an editor of, the blog binaire.blogs.lemonde.fr. He also writes novels and essays. He was a scientific curator of the exhibition "Terra Data" at the Cité des Sciences in 2017–2018.



# Automating Test Reuse for Highly Configurable Software

## An Experiment

Stefan Fischer

Institute for Software Systems Engineering  
Johannes Kepler University  
Linz, Austria  
[stefan.fischer@jku.at](mailto:stefan.fischer@jku.at)

Lukas Linsbauer

Institute for Software Systems Engineering  
Johannes Kepler University  
Linz, Austria  
[lukas.linsbauer@jku.at](mailto:lukas.linsbauer@jku.at)

Rudolf Ramler

Software Competence Center Hagenberg GmbH  
Hagenberg, Austria  
[rudolf.ramler@scch.at](mailto:rudolf.ramler@scch.at)

Alexander Egyed

Institute for Software Systems Engineering  
Johannes Kepler University  
Linz, Austria  
[alexander.egyed@jku.at](mailto:alexander.egyed@jku.at)

## ABSTRACT

Dealing with highly configurable systems is generally very complex. Hundreds of different analysis techniques have been conceived to deal with different aspects of configurable systems. One large focal point is the testing of configurable software. This is challenging due to the large number of possible configurations and because tests themselves are rarely configurable and instead built for specific configurations. Existing tests can usually not be reused on other configurations. Therefore, tests need to be adapted for the specific configuration they are supposed to test. In this paper we report on an experiment about reusing tests in a configurable system. We used manually developed tests for specific configurations of Bugzilla and investigated which of them could be reused for other configurations. Moreover, we automatically generated new test variants (by automatically reusing from existing ones) for combinations of previous configurations. Our results showed that we can directly reuse some tests for configurations which they were not intended for. Nonetheless, our automatically generated test variants generally yielded better results. When applying original tests to new configurations we found an average success rate for the tests of 81,84%. In contrast, our generated test variants achieved an average success rate of 98,72%. This is an increase of 16,88%.

## CCS CONCEPTS

- Software and its engineering → Software product lines; Software testing and debugging.

## KEYWORDS

variability, configurable software, clone-and-own, reuse, testing

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336305>

## ACM Reference Format:

Stefan Fischer, Rudolf Ramler, Lukas Linsbauer, and Alexander Egyed. 2019. Automating Test Reuse for Highly Configurable Software: An Experiment. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3336294.3336305>

## 1 INTRODUCTION

Companies develop configurable software systems to deal with the growing demand for custom tailored software products. A range of techniques have been devised for the development and maintenance of configurable software. Many large scale configurable systems, with thousands of configuration options, have been engineered. For instance, the Linux kernel has several thousands of configuration options, like supporting a wide range of different hardware from hand held devices (e.g. Android phones) to large supercomputer clusters [2].

A large number of configuration options means that there are often myriads of configurations that can be derived from the system. This variability is challenging for many tasks when working with configurable software. Not only do all the configuration options have to be considered in the development process, but also potential interactions between them. Broadly speaking, an interaction occurs when one configuration option changes the behavior associated with other options. When testing a configurable system, combinations of configuration options are of particular interest, as they may reveal undesired interactions. However, not all combinations can be tested, because the number of possible combinations usually increases exponentially with every configuration option. Krueger et al. discussed that already a system with more than 216 Boolean, non-constrained configuration options has a number of possible configurations comparable to the number of estimated atoms in the universe [11]. To handle this combinatorial explosion, commonly only subsets of possible configurations are selected for testing. For instance, Combinatorial Interaction Testing (CIT) selects configurations that cover combinations of  $n$  configuration options (therefore, often referred to as n-wise testing). A problem that still exists is that the tests for the system are themselves often not configurable [14]. In order to test different configurations, the existing tests must be adapted for each specific configuration.

The goal of this work is to investigate the possibility of reusing existing tests related to one configuration for another configuration in the context of a highly configurable software system. Our experiments are based on the widely used bug tracking system Bugzilla, which provides a large number of configuration options and can be adjusted to user needs in various ways. We implemented test variants for several different Bugzilla configurations by copying and adapting the tests from previous configurations. Such a *clone-and-own* approach is often used in practice for developing and extending related software systems [7]. Moreover, we used an automated reuse approach to generate new test variants for additional configurations that combine previously tested configuration options.

Automating the reuse of tests for configurable software can substantially reduce the effort for testing and it supports a more rigorous testing process. Krüger et al. discussed the need for automated test refactoring for the adoption of more systematic reuse approaches [12]. Thus, we applied *ECCO (Extraction and Composition for Clone-and-Own)* for automatically generating new tests from existing ones written for other configurations. *ECCO* is an approach for enhancing clone-and-own to systematically develop and maintain software variants [9]. We were able to show that the reuse support of *ECCO* is effective for automatically generating new test variants for the pairwise combination of configuration options. The approach produced 3565 executable test cases, which yielded better results than simply reusing the tests of the combined configurations in 66.34% of the new pairwise configurations and it was equally successful in 24.75%.

The remainder of this paper is structured as follows. Section 2 introduces the relevant background and Section 3 discusses the problems we aim to address and motivates our experiments. Section 4 presents the system of our study and the experiments performed with it, as well as the metrics recorded during the experiments. Sections 5 and 6 summarize the results of our experiments and discuss their implications on our research questions. Finally, Section 7 describes related work to our study and Section 8 summarizes the conclusions of our study and sketches our future work.

## 2 BACKGROUND

In this section, we discuss some of the necessary background for our work. We describe highly configurable systems, an automatic approach for reuse, and existing approaches for testing configurable software systems.

### 2.1 Highly Configurable Systems

Systems frequently offer configuration options to allow users to tailor them to their needs and preferences. These configuration options (a.k.a. features [1]) have different types of how they are expressed (e.g. Boolean options, Integers, ...) and can be realized in different forms in the system. For instance, using preprocessor directives (e.g. #IFDEFs), conditional execution (e.g. simple IFs), or build systems [14]. A large number of highly configurable systems are being maintained, ranging from just a few to thousands of configuration options (e.g. Linux kernel).

A wealth of research on highly configurable software is available in the field of Software Product Line Engineering (SPL). Software

product lines (SPLs) are families of related software systems distinguished by the set of configuration options (i.e. features) each one provides. SPLs are highly structured and they follow strict processes to deal with the contained variability. The available configuration options and dependencies between them are commonly expressed in a *variability model*.

Because SPLs typically entail a high upfront investment many practitioners use a more ad hoc approach of copying and adapting previous variants, known as *clone-and-own* [7]. However, this leads to a set of similar variants that have to be maintained separately, which becomes more difficult, the more variants being developed.

### 2.2 ECCO

In an effort to mitigate the problems associated with clone-and-own, we developed an approach called *ECCO (Extraction and Composition for Clone-and-Own)* [9, 10]. Its purpose is to support reuse in a clone-and-own context by analyzing commonalities and differences in existing variants and, subsequently, support the creation of new variants by automatically reusing relevant parts from existing variants. In a first step it *extracts* traceability information (i.e. mappings between configuration options and their implementation). The second step then allows to *compose* new variants by combining the relevant parts of the implementation using the extracted mapping information. In this paper we apply *ECCO* specifically for creating new variants of tests. First, we extract mappings between the different parts of the test source code and the configuration options that are tested by the analyzed tests. The test source code is analyzed at the level of the Abstract Syntax Tree (AST), which means that each individual element of a source code statement is considered. Then we compose new test variants by simply selecting a set of configuration options that shall be tested. The Abstract Syntax Tree (AST) of the new tests is automatically created as a *combination* of the relevant parts of the ASTs of the existing tests. Finally, the developer can manually adjust or extend the newly created variants if necessary, e.g., when the combination of existing source code parts is not sufficient to fully express new behavior of the system due to interactions or conflicts between configuration options. In context of testing, the newly generated test will likely fail when executed for the first time, indicating an unexpected and potentially erroneous interaction between configuration options.

Figure 1 shows the simplified *ECCO* workflow. Input is a set of existing variants. Each variant consists of its implementation and the information of which configuration options it encompasses. The *extraction* operation analyzes commonalities and differences in configuration options and the implementation of the variants, computes traceability information, and stores it in a repository. The *composition* operation then uses the information stored in the repository to compute the implementation of new variants given a set of desired configuration options.

### 2.3 Configurable Software Testing

There exists a substantial amount of research focusing on testing configurable software [4–6, 8]. A common thread among this research is the task to select variants for testing that are more likely to contain faulty interactions causing failures. The most prominent approaches use Combinatorial Interaction Testing (CIT) [13]. CIT

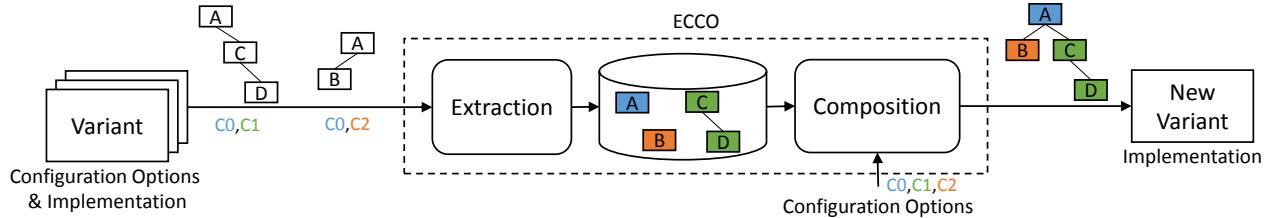


Figure 1: Simplified ECCO Workflow

techniques applied to configurable systems commonly use a variability model from which they calculate all valid  $t$ -wise configuration-option-combinations and, subsequently, covering arrays of a strength  $t$ . For instance, for  $t = 2$ , also known as pairwise testing, a CIT algorithm has to find a set of variants (i.e. covering array) to cover all combinations of two configuration option values that can be selected and which are allowed by the variability model.

### 3 PROBLEM STATEMENT

As discussed above, to test a highly configurable software system a subset of configurations has to be selected for testing. To test different configurations the tests also have to be adapted to these configurations. One solution to achieve this would be to develop the tests also as configurable software, so they can be automatically adjusted to the configurations they are supposed to test. However, in practice this is often not the case [14] as developing such tests would substantially raise development costs. Therefore, tests for a new configuration are usually created via cloning and manually adapting existing tests developed previously for a similar configuration. Following this process, practitioners end up with a set of test variants (i.e., partial clones), each to test a specific configuration.

However, covering all possible interactions that can occur in a configurable system is typically infeasible with such a manual process. Even only testing all pairwise combinations can quickly become infeasible due to combinatorics, when the tests have to be manually adapted for each configuration - not to mention the fact that these tests also have to be maintained throughout the evolution of the system [17]. In practice, testing is therefore usually focused on individual configurations (configuration options in isolation) and a few selected combinations. The limited coverage of combinations can lead to missing critical erroneous interactions between different configuration options. An approach to automatically generate tests for new configurations would help to reduce the effort of implementing and maintaining tests for a wide range of combinations and to find new interaction bugs. Given that the number of combinations is typically growing exponentially, the potential gain from using an automated approach can be huge in many projects.

Moreover, companies often use a clone-and-own process for developing new variants for their configurable system [7]. These variants are tested individually before deployment to the customers. Tests are reused from previous variants and are also adapted in a clone-and-own manner [12]. There are many benefits for migrating from clone-and-own to a more systematic approach on a reusable platform, like reduction in maintenance costs. A barrier preventing

such a migration is often the fear of introducing new bugs during the migration [12]. Being able to automatically reuse tests from previous variants could help with this issue and it would allow to ensure that the system still behaves as expected after migration.

These practical problems motivated us to investigate systematic and automated reuse for software tests of configurable systems and to perform the experiments discussed in this paper. In particular we aim to answer the following research questions:

**RQ1: To what degree can tests from configurations be reused directly?** We analyze how many of the existing tests can be directly applied for testing other configurations and in how many cases modifications are required. Hence, this question allows us to assess the manual effort that would be required to adapt existing tests for different configurations.

**RQ2: To what degree can we automatically generate test suites for new configurations from existing tests?** The main goal of our experiments is to determine whether we can automatically compose test variants for new, previously untested configurations by reusing parts of the source code of existing tests. Therefore we investigate the use of the *ECCO* tool support for automatically composing such tests.

### 4 EXPERIMENT DESIGN

In this section, we discuss the methodology of our experiment. We start with explaining the system under test and the existing tests we have developed, followed by the setup used for our experiments, and the metrics measured during these experiments.

#### 4.1 System Under Test

The configurable system we used in our experiments is the widely used, open-source bug tracker Bugzilla. Specifically, we used the Virtual Bugzilla Server (version 3.4) provided by ALM Works<sup>1</sup>. This is a virtual machine image containing a ready-to-use setup of the Bugzilla 3.4 Web application, an Apache Web server, and a MySQL database running on Debian Linux. Bugzilla is a Web-based application, so the front-end (user interface) of the Bugzilla server is accessed using a Web browser.

We initially implemented a suite of 34 automated test cases exercising the main functionality of Bugzilla via the Web front-end (e.g., submitting a bug report, searching and updating a report, changing the bug status). The tests are written in Java and use Selenium<sup>2</sup> to control the Chrome Web browser to interact with Bugzilla. The tests run on the default configuration of Bugzilla. Subsequently,

<sup>1</sup><https://almworks.com/archive/vbs>

<sup>2</sup><https://www.seleniumhq.org/>

we identified a range of different configuration options that can be used to change the default behavior of Bugzilla. We selected a diverse set of fifteen different options resulting in configuration changes that are directly observable in the Web front-end, in the navigation structure, or in the bug tracking workflow of Bugzilla. Thus, these options can be expected to impact our existing set of tests and make adaptations necessary in order to run them after a configuration change. Furthermore, some of the configuration options are expected to result in conflicts when activated in combination. We created tests for each of the additional configurations by manually performing clone-and-own starting from the test cases for the default configuration.

Config	Tests	Description
C00	34	Default configuration of Bugzilla
C01	36	Enable status white board field for optional comments
C02	34	Disable priority selection on bug report submission
C03	33	Allow using empty values in bug search form
C04	35	Add an additional product to organize bug reports
C05	36	Add an additional component to a product
C06	36	Add an additional version to a product
C07	34	Configure bug status workflow to a minimum set of states
C08	34	Configure bug status workflow to a different entry state
C09	35	Require descriptions on creating a new bug entry
C10	34	Require descriptions on all bug status changes
C11	35	Require resolution description on setting a bug to resolved
C12	35	Enforce a comment when a bug is marked as duplicate
C13	35	Enforce dependencies to be resolved before bug can be fixed
C14	34	Set the default bug status of duplicates to verified
C15	34	Set the default bug status of duplicates to closed

Table 1: Configurations and Number of Tests

We list the 16 configurations of Bugzilla used in our experiments in Table 1 along with the number of test cases developed in each of the variants and a description of the impact of changing a specific configuration option. The changes in the different configurations range from simply adding an optional comment field (C01) to completely changing the bug workflow and the states that can be assigned to a bug (C07 and C08). Some of these configuration options are related to the same functionality and we therefore expect conflicts if they are set simultaneously with one another. For instance, C07 and C08 both change the bug workflow and therefore they cannot both be configured at the same time. Configuration C12 can only be used when duplicates are allowed. A conflicting dependency also exists between configurations C14 and C15, because they both change the default status of duplicates to different states and therefore, they cannot be set simultaneously. Furthermore, we might run into another conflict if any of the two is activated together with C07, because they change a bug status that might no longer be allowed in C07.

Each of the test suite variants that were developed to test a specific configuration consists of a (1) *Test Set Up* that configures the Bugzilla server accordingly (i.e. activates the configuration and resets it to the default configuration after the tests were executed), (2) *Test Cases* that exercise the functionality of Bugzilla in various ways, and (3) *Page Objects* that use Selenium to access Bugzilla through its Web front-end. Figure 2 sketches the test execution cycle of one of the test variants targeting a specific configuration. All of the parts are implemented in Java and each test variant is an

independent Maven<sup>3</sup> project that has been created by cloning and modifying the tests for the default configuration (C00). We use the tool Maven Invoker<sup>4</sup> to automatically execute all test variants. As depicted in Figure 2, we first call the *Test Set Up* to set Bugzilla to the desired configuration. Next, we use the JUnit test runner to execute the *Test Cases*, which call methods provided by the *Page Objects*. These are Java objects that use Selenium to interact with the Web pages realizing the Bugzilla front-end and to verify the expected outcome. Finally, we use the *Test Set Up* to reset the configuration back to its default in order to provide a clean basis for running other test variants using the same process.

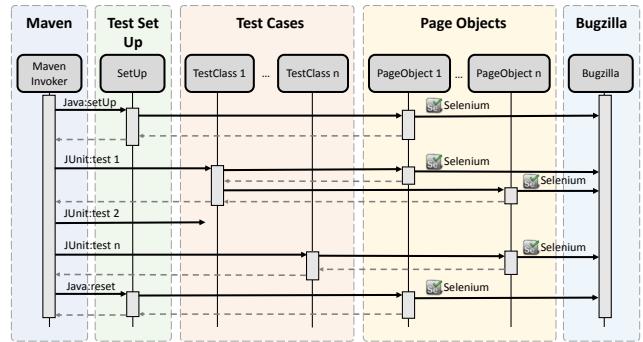


Figure 2: Sequence of Executing a Test Variant

## 4.2 Composing New Test Variants

We applied *ECCO* to create new test variants from existing tests. We generated the tests for new configurations that are combinations of configuration options covered individually by the existing tests. We did not adjust *ECCO* for this experiment. It was used as described in Section 2.2 on the existing test variants along with the covered configuration options as input.

Figure 3 shows code snippets of three different manually developed test variants, testing the configurations C00, C01, and C06 respectively. The configuration C00 is the variant testing the default configuration of Bugzilla. Configuration C01 adds an optional text field for commenting on the status of a bug, that is accessed in Line 24 in test variant T01. Although test T00 was written for the default configuration C00 it can still run successfully on C01, because the text field is optional and not accessed by the test. Test T01 can not successfully run on configuration C00 and will cause an error in Line 24, since the text field does not exist in this configuration.

Configuration C06 adds another product version to the Bugzilla default configuration, which then overrides the default selection of the version labeled *unspecified*. All test variants assert that the version of the created bug entry is *unspecified* (Lines 9, 22, and 38 respectively). However, if we execute the tests T00 or T01 on configuration C06 we would get a failure from this assertion, because the default version has been overridden and the new version is selected instead. The test variant T06 was therefore adapted by adding Line 33 that explicitly sets the version to *unspecified*.

<sup>3</sup><https://maven.apache.org/>

<sup>4</sup><https://maven.apache.org/shared/maven-invoker/>

Variant T00 (BASE):

```

1 class CreateNewBugTest {
2     public void testCreateBugDefaultValues() {
3         ...
4         createBug.setSummary(summary);
5         BugCreatedPage created=createBug.commitBug();
6         ...
7         EditBugPage editBug=created.gotoCreatedBugPage();
8         assertEquals(summary, editBug.getSummary());
9         assertEquals("unspecified",editBug.getVersion());
10        ...
11    }
12 }
```

Variant T01 (BASE + USESTATUSWHITEBOARD):

```

14 class CreateNewBugTest {
15     public void testCreateBugDefaultValues() {
16         ...
17         createBug.setSummary(summary);
18         BugCreatedPage created=createBug.commitBug();
19         ...
20         EditBugPage editBug=created.gotoCreatedBugPage();
21         assertEquals(summary, editBug.getSummary());
22         assertEquals("unspecified",editBug.getVersion());
23         ...
24         assertEquals("",editBug.getStatusWhiteboard());
25         ...
26    }
27 }
```

Variant T06 (BASE + ADDVERSION):

```

29 class CreateNewBugTest {
30     public void testCreateBugDefaultValues() {
31         ...
32         createBug.setSummary(summary);
33         createBug.setVersion("unspecified");
34         BugCreatedPage created=createBug.commitBug();
35         ...
36         EditBugPage editBug=created.gotoCreatedBugPage();
37         assertEquals(summary, editBug.getSummary());
38         assertEquals("unspecified",editBug.getVersion());
39         ...
40    }
41 }
```

Figure 3: Source Code Snippets of Bugzilla Tests

We used the test variant for the default configuration C00 and the different variants created for the additional fifteen configurations C01-C15 as input for *ECCO*. Furthermore, we also provided the configuration option tested by each of the variants to allow *ECCO* to establish links between configuration options and the related source code parts of the tests. Based on the extracted knowledge, *ECCO* generates the source code of the tests for a new configuration, which is specified in terms of the activated configuration options.

Figure 4 shows code snippets of a test generated with *ECCO* for the combination of the configurations C01 and C06. This new test variant contains the code for the optional status text field in Line 53 and for setting the added bug version in Line 46. Therefore, this new test can be executed on the combined configuration with USESTATUSWHITEBOARD and ADDVERSION activated. In contrast, the tests T00 and T01 would fail on this combination due to the change caused by the added version. Test T06 would still pass, but it does not assert that the optional status text field has been activated. The test variant generated with *ECCO* also executes the assertion for the optional text field and therefore achieves higher code coverage.

Variant T01-06 (BASE + USESTATUSWHITEBOARD + ADDVERSION):

```

42 class CreateNewBugTest {
43     public void testCreateBugDefaultValues() {
44         ...
45         createBug.setSummary(summary);
46         createBug.setVersion("unspecified");
47         BugCreatedPage created=createBug.commitBug();
48         ...
49         EditBugPage editBug=created.gotoCreatedBugPage();
50         assertEquals(summary, editBug.getSummary());
51         assertEquals("unspecified", editBug.getVersion());
52         ...
53         assertEquals("",editBug.getStatusWhiteboard());
54         ...
55    }
56 }
```

Figure 4: Source Code Snippets of Composed Test

### 4.3 Experiment Execution

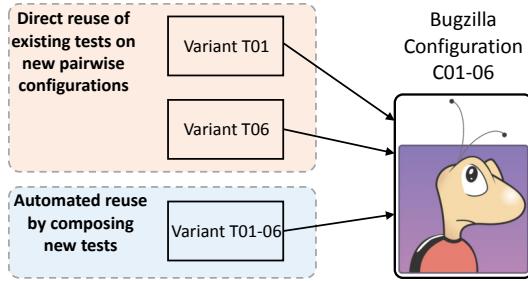
We performed several different experiments to answer our research questions stated above.

**Direct reuse of existing tests on other configurations:** To answer *RQ1*, we investigated how many of the existing tests of the original variants could be directly reused for testing other configurations. We first executed all tests on the individual configuration they were created for to ensure they work correctly. Then we executed each of the tests also on all other configurations to find out how much the individual configurations influence the test runs. From the results we analyzed to what degree the tests are influenced by the different configurations.

**Direct reuse of existing tests on new pairwise configurations:** We created new configurations by building pairwise combinations of existing configurations, i.e., by activating the Bugzill options related to two individual configurations simultaneously. Then, we executed the tests for each of the two configurations to evaluate the reuse of the existing tests on these new pairwise configurations. To activate the configuration options in Bugzilla, we used the existing setup code from both of the involved test variants and executed them in sequence, so both options would be set. In order to mitigate the possibility that the setup of the first configuration influences the other one (e.g., one masking the other and corrupting the outcome), we performed the experiment twice and changed the order in which the two setups were executed. This allowed us to assess to what degree the original variants can be directly reused on pairwise configurations.

**Automated reuse by composing new tests:** To address *RQ2*, we also executed newly composed tests on the pairwise combinations of existing configurations. We applied *ECCO* to generate new test variants for all possible combinations of any two existing configurations. The goal of this experiment was to assess the usefulness of automatically composing new tests from existing tests for new configurations. Figure 5 illustrates the experiment on pairwise configurations for the example of testing configuration C01-06. The pairwise configuration C01-06 represents the combination of the individual configurations C01 and C06. Instead of testing this new configuration with the existing tests T01 (developed for the configuration C01) and T06 (developed for C06), Please note that *ECCO* is

deterministic and therefore we only had to perform this experiment once.



**Figure 5: Example Experiments on Pairwise Configurations**

**Automatic reuse of setup code:** We can use *ECCO* not only to generate test code for the new configurations but also for generating the setup and reset code to configure Bugzilla accordingly. In the previous experiment, we executed the setup of each individual configuration in sequence to activate all required configuration options. To investigate if the *ECCO* approach is also applicable for composing setup code, we repeated the previous experiment, using the *ECCO* generated setup and reset code instead of the original ones. We compared the results to those from the previous experiment in order to assess how well the *ECCO* setup code worked.

All test variants used in the experiments, original ones as well as those generated with *ECCO*, were realized as separate Maven projects. We used the Maven Surefire Plugin<sup>5</sup> to generate test reports and the JaCoCo<sup>6</sup> Maven Plugin to record code coverage.

#### 4.4 Metrics

Next, we will discuss the metrics we recorded for our experiments. From the test reports we can extract the first set of metrics.

##### Test Result Metrics.

- **Number of Test Cases Tests.** The number of test cases that exists for a variant.
- **Number of Successful Tests Succ.** The number of test cases that were executed in a variant without any problem (i.e. no failures or errors).
- **Test Success Rate SuccessRate.** The rate in which test cases could be executed without problem (i.e. no failures or errors).

$$\text{SuccessRate} = \text{Succ}/\text{Tests}$$

Furthermore, as the existing tests may still pass but yield less coverage than the composed ones, we also analyze and compare the coverage achieved by the tests. However, we were not able to measure the actual coverage of the Bugzilla code. Instead, we measured the coverage of the Java code for the page objects of the executed variants. Our reasoning for doing this was that each page object represents an actual page of Bugzilla, and the code that was executed uses parts of the page. Therefore, we argue that coverage of the page object should logically be correlated with the coverage of Bugzilla itself. The coverage report from JaCoCo includes lines, methods, and classes that have been executed during

<sup>5</sup><https://maven.apache.org/surefire/maven-surefire-plugin/>

<sup>6</sup><https://www.eclemma.org/jacoco/>

testing. However, the line coverage metric is influenced by the formatting (i.e. a statement can be in one line or split into several lines). Because *ECCO* may format some statements different than they were in the original variants we computed statement coverage instead, which allows a better comparison. We did this by iterating over the Abstract Syntax Tree (AST) (generated with the Eclipse Java development tools (JDT)) and checking the JaCoCo report for each statement if the corresponding line was executed. Moreover, from this AST we computed the number of statements that exist in each variant and over all variants combined.

##### Coverage Metrics.

- **Number of Statements FullCount.** The number of unique statements that exist combined over all variants.
- **Number of Executed Statements VarCovered.** The number of statements that have been executed when testing a variant.
- **Statement Coverage on all code OverallCoverage.** Coverage of statements in an individual variant in relation to all statements of all variants.

$$\text{OverallCoverage} = \text{VarCovered}/\text{FullCount}$$

Therefore, *OverallCoverage* is the proportion of Bugzilla that we can access with our page objects and which is executed during testing.

## 5 RESULTS

In this section, we present the results of our experiments.

### 5.1 Direct Reuse of Existing Tests on Other Configurations

First, we executed all test variants on all configurations. In Figure 6 we depict the *SuccessRate* at which test cases from existing test suite variants (columns) could be applied to existing configurations (rows). As is expected, the diagonal is filled with the values 1.0, meaning 100% of the test cases could be applied to the configurations they were developed for. Moreover, some variants can apply all their test cases to some other configurations, for example tests from C10 (i.e. T10) can be applied to six configurations besides C10 itself. Most of the variants can run a fairly high number of their test cases on other configurations (between 70% and 90%). The exception for this are the configurations C04 and C05, on which only the most basic tests from other configurations could be executed. Similarly, the test suite for configuration C04 (i.e. T04) could not run many of its tests on other configurations than C04 itself.

### 5.2 Direct Reuse of Existing Tests on New Pairwise Configurations

Next, we executed the tests on pairwise configurations. There are 105 possible pairs in total ( $\binom{15}{2} = 105$ ). The order in which we executed the setup made no difference for the results of most of the configurations. For configuration C14-15 we found a difference in the *SuccessRate* depending on the order, due to the expected conflict between the two configurations. The *SuccessRates* did an exact flip with the order in which the configurations was set up and we included the results, because there was effectively no difference in the numbers. Furthermore, we also discovered that the combined

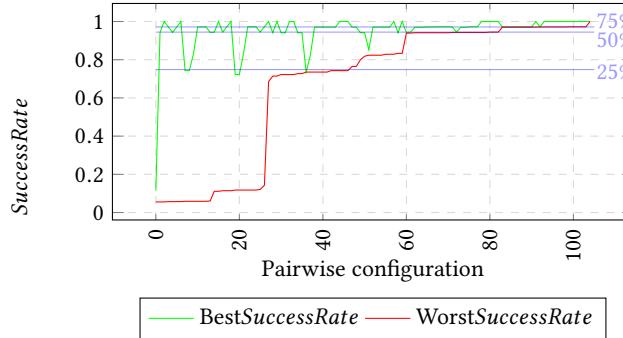
	T00	T01	T02	T03	T04	T05	T06	T07	T08	T09	T10	T11	T12	T13	T14	T15
C00	1,00	0,81	1,00	0,84	0,06	0,89	0,89	0,85	0,91	0,97	1,00	0,89	0,97	0,97	0,97	0,97
C01	1,00	1,00	1,00	0,84	0,06	0,89	0,89	0,85	0,91	0,97	1,00	0,89	0,97	0,97	0,97	0,97
C02	0,97	0,78	1,00	0,81	0,06	0,89	0,89	0,82	0,91	0,94	0,97	0,86	0,94	0,94	0,94	0,94
C03	0,94	0,75	0,94	1,00	0,06	0,83	0,83	0,79	0,85	0,91	0,94	0,83	0,91	0,91	0,91	0,91
C04	0,06	0,06	0,06	0,06	1,00	0,06	0,06	0,06	0,06	0,06	0,06	0,06	0,06	0,06	0,06	0,06
C05	0,12	0,11	0,12	0,12	0,06	1,00	0,11	0,12	0,12	0,11	0,12	0,11	0,11	0,11	0,12	0,12
C06	0,94	0,78	0,94	0,78	0,06	0,89	1,00	0,79	0,91	0,91	0,94	0,83	0,91	0,91	0,91	0,91
C07	0,74	0,67	0,74	0,69	0,06	0,67	0,67	1,00	0,71	0,71	0,74	0,71	0,71	0,74	0,71	0,71
C08	0,82	0,69	0,82	0,69	0,06	0,78	0,81	0,85	1,00	0,80	0,82	0,80	0,80	0,79	0,79	0,79
C09	0,97	0,81	0,97	0,81	0,06	0,89	0,89	0,82	0,91	1,00	1,00	0,86	0,94	0,94	0,94	0,94
C10	0,74	0,69	0,74	0,69	0,06	0,67	0,67	0,74	0,71	0,77	1,00	0,69	0,74	0,71	0,74	0,74
C11	0,94	0,75	0,94	0,78	0,06	0,83	0,83	0,85	0,85	0,91	1,00	1,00	0,91	0,94	0,91	0,91
C12	0,97	0,78	0,97	0,81	0,06	0,86	0,86	0,82	0,88	0,94	1,00	0,86	1,00	0,94	0,97	0,97
C13	1,00	0,81	1,00	0,84	0,06	0,89	0,89	0,85	0,91	0,97	1,00	0,89	0,97	1,00	0,97	0,97
C14	0,97	0,78	0,97	0,81	0,06	0,86	0,86	0,82	0,88	0,94	0,97	0,86	0,94	0,94	1,00	0,97
C15	0,97	0,78	0,97	0,81	0,06	0,86	0,86	0,82	0,88	0,94	0,97	0,86	0,94	0,94	0,97	0,97

**Figure 6: SuccessRate for Reusing the Test Cases of each Test Variant (Columns) on all Configurations (Rows)**

setup and reset did not work for four configuration pairs (C07-09, C07-10, C08-09, and C08-10) when running our experiments. We were able to run the setup for these variants in at least one order so we were able to retrieve the data for the experiment, but we had to manually reset the virtual machine image of Bugzilla to re-establish the default configuration.

Figure 7 depicts the *SuccessRate* for executing the original variants on the 105 pairwise configurations. We executed the two variants corresponding to the two combined configurations. Subsequently, we classified them in the best and the worst of the two variants and printed them sorted by the *WorstSuccessRate*. Moreover, we depict the quantiles for 25%, 50%, and 75% of the entire *SuccessRate* data.

For the majority of the 105 configurations none of the original test variants could be applied successfully (74 times). In 30 cases, we were able to reuse one test variant completely and in one case (C01-13) both of them worked.



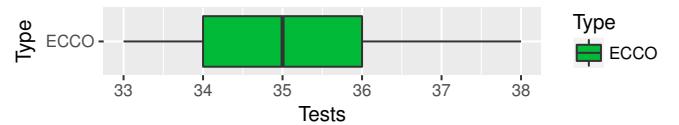
**Figure 7: SuccessRate of Original Variants on Pairwise Combinations**

### 5.3 Automatic Reuse by Composing New Tests

Next, we generated the pairwise test variants using *ECCO* for each of the 105 pairwise configurations. It took 33.1 seconds to extract the mapping information from the 16 initial variants and 49.3 seconds to generate all 105 new variants. We used a system with an Intel

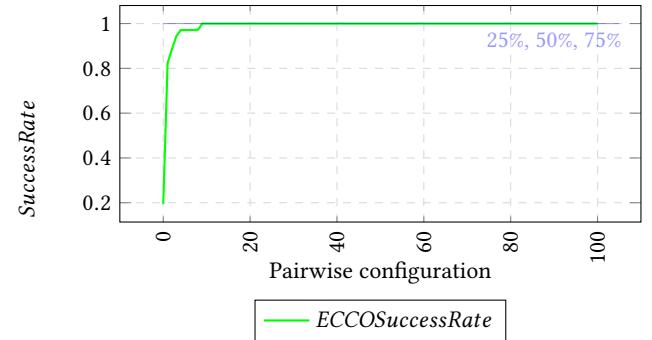
i7-3610QM CPU @2.3 GHz and 16GB of RAM. Four out of these 105 variants resulted in a compiler error and could therefore not be executed (T04-05, T04-09, T09-11, and T09-12). These errors occurred at positions where the AST merge lead to merge conflicts that *ECCO* can not automatically decide. For instance, when two different return statements appear at the end of a method or when the same variable is defined in a method twice due to the merge. We executed the tests for the remaining 101 variants and computed our metrics. The order of the setup did not affect the results of the tests generated by *ECCO*.

Figure 8 shows the number of test cases part of the 101 working *ECCO* generated variants. They range from 33 to 38 test cases, whereas the number of test cases for the original variants only ranged from 33 to 36 test cases. These results can be expected as *ECCO* merged test cases from two different variants into one variant.



**Figure 8: Number of Test Cases (*Tests*) in the *ECCO* Variants**

Figure 9 depicts the *SuccessRate* of the 101 working generated *ECCO* variants on the pairwise configurations. We observed a very high *SuccessRate* for the *ECCO* generated test variants. In fact, in 92 of the 101 variants all tests passed successfully (i.e. *SuccessRate* = 1.0), which is also why all the quantiles are at 1.0.



**Figure 9: SuccessRate of *ECCO* Variants on Pairwise Combinations**

We compared these results with the ones from using the original variants on the pairwise combinations. Table 2 shows the number of times that our with *ECCO* generated tests could all be executed successfully compared to when executing the original test variants on the pair-wise configurations. We can see that for the majority of the 101 configurations none of the original test variants could be applied successfully (70 times). In contrast, the *ECCO* generated test variants passed successfully for 67 of these 70 configurations. In 30 cases it was possible to reuse one test variant completely, and

		Original Variants			Total
		Successful			
ECCO Pair Variants	Success	None	One	Both	Total
		67	24	1	92
Total	Fail	3	6	0	9
Total		70	30	1	101

**Table 2: Contingency Table of Successfully Passing Tests for ECCO Variants vs. Original Variants**

in only one configuration we were able to apply both original test variants.

For some configurations our results showed a very low *SuccessRate*, as we can see in the outliers at the start in Figure 9. Moreover, Table 2 shows that for three configurations none of the variants worked without problems. These three configurations are also the ones where our *ECCO* variants performed the worst in terms of the *SuccessRate*. The worst case in the *ECCO SuccessRate* stems from configuration C05-10 (i.e. the combination of C05 and C10), with a *SuccessRate* of only 19.4%. We found that C05 changes the process for many tested use cases of Bugzilla and, therefore, most tests failed in the pairwise combinations containing C05 and in the *direct reuse* results shown in Figure 6. Nonetheless, the *ECCO* variants for combinations with C05 worked without a problem. The combination with C10 seems to work specifically poorly, because it also requires a comment on all bug status changes that the tests from T05 do not include.

The next configuration that did not work with any variants and which resulted in the second worst *SuccessRate* for the generated *ECCO* variant was C07-08. However, we expected conflicts for the combination of these configurations, as discussed in Section 4.1.

The third and final configuration that each applied variant encountered problems on was C07-11. Test from T07 failed because C11 requires a comment on bug resolution change that is not implemented in the tests, and T11 failed because C07 restricts the Bugzilla workflow and bug status values that are possible and therefore bugs have another status then expected by the tests.

Other interesting results were for combinations that we actually expected conflicts. For instance, we found that for the combination of C09 and C10 we indeed found a conflict when applying the tests generated with *ECCO* (i.e. 1 failed test in T09). However, the tests of T10 worked on the combination without a problem, because C10 requires comments on all bug status changes including the one required by C09. Moreover, we expected conflicts for configurations C07-14 and C07-15, but found the *ECCO* tests worked without problems, because the status changes tested for duplications still worked in the minimal workflow configuration. In contrast, all original variants encountered some problems during testing, like T07 that expected a different status for bug duplicates and the rest did not work on the minimal workflow. Finally, we also expected a conflict for combining C14 and C15 and indeed found that we can not configure both of them at the same time, because they configure the same configuration option. Hence, the *ECCO* test variant did not work and which one of the original variants that worked depended on the order of configuration (i.e. for the configuration that was configured last the tests in the corresponding variant worked).

## 5.4 Automatic Reuse of Setup Code

Next, we performed the experiment again with the setup code generated by *ECCO*. For most configurations and test variants the results did not change compared to the previous experiment above. The four variants that could not setup/reset the configuration correctly also could not reset Bugzilla to the default configuration with the *ECCO* generated setup code. Therefore, we had to stop the virtual machine running Bugzilla and reset the image manually, before continuing the experiments in the same way we did in the previous experiment.

When further investigating the differences in our results we found two configurations that behaved slightly different with the *ECCO* setup. The most severe differences in the results occurred in variant T07-08 that already confirmed our expected conflicts in the previous experiment. However, with the *ECCO* setup the results are even worse and each variant caused 32 errors out of 34 test cases, which translates to a *SuccessRate* of 5.9%. The second configuration for which we found differences in our test results was C09-10 where variant T09-10 caused one error in the previous experiment. Surprisingly, variant T09-10 works without any problem when we used the *ECCO* setup to establish configuration C09-10. We investigated why this is the case and found that the setup code in variant T09 is a subset of the code in T10, and therefore the merged variant T09-10 had the equal setup code as T10. This is also why tests from T10 work on configuration C09. Finally, for configurations C14-15 we found in the previous experiment that the order of the setup mattered for the test outcomes, and therefore the results for this experiment matched only the results of the setup order that matched the order that *ECCO* generated.

## 6 DISCUSSION

In this section we discuss the implications of the results on our research questions.

**RQ1: To what degree can tests from configurations be reused directly?** In our first experiment we found that some test variants work without any failures or errors on other configurations (see Figure 6). This was the case because some configurations enabled a subset of configuration options on other configurations, like C09 where a comment is only required in a subset of the cases of C10 and therefore the tests of C10 (i.e. T10) also work on C09. For other configurations (i.e. C04 and C05) we found that only some of the basic tests worked and most others failed, because these configurations change many aspects of the main use cases of Bugzilla that even involve new Web pages that the other tests were not designed to interact with. However, the majority of the remaining variants could execute between 67% and 97% of their test cases on other configurations without problems.

The second experiment showed that the *direct reuse* works in 31 of the 105 configurations for at least one variant. However, this leaves the majority of test variants to be adapted in terms of fixing failing tests, which requires considerable manual effort.

**RQ2: To what degree can we automatically generate test suites for new configurations from existing tests?** Our results suggest that *ECCO* is useful for automatically generating new test variants. The data of our *automatic reuse* experiment showed a significantly higher *SuccessRate* for test variants generated with

*ECCO* compared to directly reusing existing variants on the new configurations. We measured an average *SuccessRate* of 98,72% for tests we generated with *ECCO*, compared to a *SuccessRate* of 81,84% for using the two original test variants. Even if we always select the original variants with the highest *SuccessRate* for every pairwise configuration, the average *SuccessRate* would be less (95.8%). However, the information required to make this selection for a specific configuration options is unknown before execution. If we would instead always choose the worse of the two variants, the average *SuccessRate* drops to 67.9%.

Figure 10 depicts the rates in which tests cases were successful for the generated tests (i.e. *ECCO*), and for the two variants testing the two configurations that were merged. We can see that the test variants generated with *ECCO* have a significantly higher *SuccessRate* than any of the other variants. For 31 configurations we were able to successfully execute an existing test variant of the previous configurations, which means 32 initial variants could be applied (since for one configuration we could reuse both variants fully). We confirmed the statistical significance of the results using the Wilcoxon-Rank-sum test ( $p\text{-value}: 2.2 \exp(-16)$ ) [16]. Additionally, we computed the effect size measure  $\hat{A}_{12} : 0.886$ , which means our generated test variants lead to a higher *SuccessRate* than the original two variants in 88.6% of the cases.

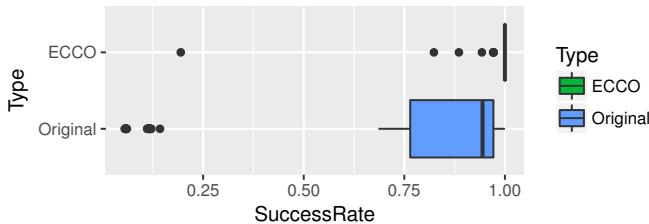


Figure 10: SuccessRate of *ECCO* Tests vs. Original Tests

Figure 11 depicts the *OverallCoverage* for the different test variants. We found that generally the *ECCO* tests have higher coverage in most cases and, therefore, they executed more of the code of our page objects. However, since *ECCO* merges code from original variants we would expect the generated variants to contain more code to execute than the original variants.

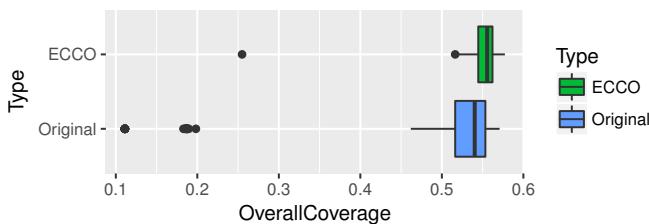


Figure 11: OverallCoverage of *ECCO* Tests vs. Original Tests

Finally, we investigated the relations between *SuccessRate* and *OverallCoverage* of our results. To do this we classified the results depending on the *ECCO* variants results compared to the results for

the metrics of other variants. In Table 3 we show the contingency table of this comparison. We can see that for 34 configurations the *ECCO* variants were best in both metrics. Moreover, for the configurations in which the *ECCO* variants *SuccessRate* was equal to or worse than for other variants, the *ECCO* variant still had the highest *OverallCoverage* in the majority of cases.

		<i>OverallCoverage</i>					Total
		<i>ECCO</i>	<i>ECCO+1</i>	<i>ECCO+2</i>	<i>ECCO-1</i>	<i>ECCO-2</i>	
<i>SuccessRate</i>	<i>ECCO</i>	34	20	0	12	1	67
	<i>ECCO+1</i>	18	4	0	2	0	24
	<i>ECCO+2</i>	1	0	0	0	0	1
	<i>ECCO-1</i>	6	0	2	1	0	9
	<i>ECCO-2</i>	0	0	0	0	0	0
	Total	59	24	2	15	1	101

*ECCO*: *ECCO* results were the best,  
*ECCO+X*: *ECCO* results were equally best with 1 or 2 other variants,  
*ECCO-X*: *ECCO* results were worse than 1 or 2 other variants

Table 3: Contingency Table *SuccessRate* vs. *OverallCoverage*

These results support the general usefulness of *ECCO* to automatically generate new test variants. Moreover, we found that *ECCO* can also be used to generate the test setup code of our test variants. Reducing the effort for reusing existing test code for new configurations is beneficial for testing highly configurable software. Furthermore, *ECCO* could be used for refactoring tests when moving variants that were developed using clone-and-own to a platform for systematic reuse.

## 6.1 Limitations

In our experiments we only measured the rate in which test cases succeeded and the code coverage. We did not have any fault data, so we were not able to investigate if the tests would actually be able to discover faults in the system. To further study the usefulness of an automated reuse approach for tests and to evaluate the generated test quality, measuring the fault detection capabilities is the next logical step. However, for now this has to remain an item on our future work agenda. Nonetheless, demonstrating that we can generate working test variants for new configurations is an important step into the direction of automated test reuse.

Another limitation of our work we want to point out is that we only generate test variants for configurations which are pairwise combinations of previously tested configurations. We do not generate test suites for entirely new configurations. With *ECCO* we can automatically reuse tests by mapping configuration options to test code that already exists, but the approach cannot be used to generate entirely new test code.

## 6.2 Threats to Validity

**External validity:** Our study only includes one configurable system. Studies on more systems are required to determine the degree to which results may be generalized. However, the system that we used is well known and should be representative for this type of configurable systems. A possible source for bias might be the configuration options we selected to use in our experiment. This choice was based on selecting arbitrary options from the Bugzilla configuration pages that have an observable impact on the user interface. Another possible source for bias might be that the tests were also created by the authors. We developed the tests for the

default configuration and then for the other 15 configurations using a clone-and-own process, all before the experiments. We did not alter the tests at all for our experiments, so they are more realistic and even led to compiler errors in four of the variants generated by *ECCO*.

**Internal validity:** We required several tools to perform the experiments and for data analysis. Errors in these tools might bias our results. To reduce this possibility, we validated all used tools and our code on smaller examples and subsets of the data. Another possible source for bias might come from the automatic setup of the configurations. To reduce this possibility, we randomly checked the configurations in Bugzilla and ensured it was configured as intended. Furthermore, we performed the experiments on pairwise configurations with the setup in both orders and with the setup generated by *ECCO*. To proof the applicability of *ECCO* for our test code we used it to reconstruct the original variants. Hence, we used all 16 variants as input for *ECCO* and regenerate all of them. We compared the Abstract Syntax Tree (AST) of the original variants with *ECCO*'s reconstruction of the variants and found no difference. Moreover, we executed all the tests on the reconstructed variants like in our first *direct reuse* experiment and compared the results and found no difference in the test results. Similarly, we found no difference in the coverage data we recorded. These results confirm the basic usefulness of *ECCO* for our experiment and showed that it successfully can identify parts specific to configuration options from the initial variants.

**Construct validity:** We measured test success and code coverage on the Java page objects. Instead of only measuring which tests run without problem it would also be interesting to test for faults in the system and compare which tests are able to detect faults in different configurations. However, we did not have any faults for our system and were also not able to perform mutation testing on the virtual machine that runs the tested system. Moreover, we measured the code coverage only on the Java page objects, because we could not measure coverage within the virtual machine. We argue that the coverage of the page objects is likely to correlate with the coverage of the corresponding Bugzilla page, but this might be a source for bias in our results.

## 7 RELATED WORK

Cohen et al. performed an experiment to show the effect of executing tests for different configurations of a highly configurable system [3]. They found small differences in fault detection and code coverage across configurations. This experiment is similar to our first experiment for *direct reuse*, were we also found that many test cases still worked on different configurations. However, we did not have fault data available, nor could we inject mutants like Cohen et al. did in their experiments, which is a limitation of our work. The main difference of this work and the work from Cohen et al. is that our main goal was to assess our capabilities to automatically generate new variants to test combinations of previous configurations, which was out of the scope of the experiments of Cohen et al.

Ramler et al. reported their experience for automatically reusing tests across configurations and versions to increase code coverage [15]. They were able to increase coverage by directly reusing test cases from other configurations. We found similar results in our

first experiment for *direct reuse* that showed that we can reuse some test cases for one configuration on other configurations without problems. Additionally, we performed experiments for automatically generating new test variants.

As we have mentioned before, Krüger et al. discuss the need for automatic refactoring of tests to reduce barriers of moving from variants developed with a clone-and-own process to a more systematic SPL platform [12]. They discuss challenges linked to such a refactoring and outline their own ideas for such a refactoring approach. Our experiments support the usefulness of using *ECCO* for reusing tests and we argue that with *ECCO* we can address several of the challenges discussed by Krüger et al.

## 8 CONCLUSIONS AND FUTURE WORK

In this paper we performed experiments on the reusability across configurations of a highly configurable software system. Furthermore, we used an approach for automatic reuse to generate tests for new configurations by reusing previously developed test variants. Our experiments showed that for most configurations a large proportion of around 70% to, in some cases, 100% of tests cases could be applied to other configurations without problems. The main goal of our study was to assess the usefulness of automatically generated test variants, for which we found an average success rate of 98.7%, compared to 81.8% when directly reusing previous variants. These results suggest a considerable advantage of our approach to automatically generate tests over the direct reuse approach, which requires additional manual effort for adapting the failing tests.

However, more experiments are required to confirm these findings. In our future work, first, we plan to use also other configurable systems to replicate our results. Ideally, these systems would have fault data available or allow to use mutation testing, so we could also assess the fault detection capabilities of different test variants. Additionally, we would be interested in performing further experiments with new configurations and other variants (e.g. 3-wise combinations). Finally, we plan to investigate if we can use the results from testing different configuration combinations to infer the existence of unknown interactions among configuration options.

## ACKNOWLEDGMENTS

The research reported in this paper has been supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry for Digital and Economic Affairs, and the Province of Upper Austria in the frame of the COMET center SCCH, grant no. FFG-865891. Furthermore, this research was in part funded by the JKU Linz Institute of Technology (LIT) by the state of Upper Austria, grant no. LIT-2016-2-SEE-019.

## REFERENCES

- [1] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a feature?: a qualitative study of features in industrial software product lines. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20–24, 2015*, Douglas C. Schmidt (Ed.). ACM, 16–25. <https://doi.org/10.1145/2791060.2791108>
- [2] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. Software Eng.* 39, 12 (2013), 1611–1640. <https://doi.org/10.1109/TSE.2013.34>

- [3] Myra B. Cohen, Joshua Snyder, and Gregg Rothermel. 2006. Testing across configurations: implications for combinatorial testing. *ACM SIGSOFT Software Engineering Notes* 31, 6 (2006), 1–9. <https://doi.org/10.1145/1218776.1218785>
- [4] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John D. McGregor, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. 2011. A systematic mapping study of software product lines testing. *Information & Software Technology* 53, 5 (2011), 407–423.
- [5] Ivan do Carmo Machado, John D. McGregor, Yguratará Cerqueira Cavalcanti, and Eduardo Santana de Almeida. 2014. On strategies for testing software product lines: A systematic literature review. *Information & Software Technology* 56, 10 (2014), 1183–1199.
- [6] Ivan do Carmo Machado, John D. McGregor, and Eduardo Santana de Almeida. 2012. Strategies for testing products in software product lines. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–8.
- [7] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5–8, 2013*, Anthony Cleve, Filippo Ricca, and Maura Cerioli (Eds.). IEEE Computer Society, 25–34. <https://doi.org/10.1109/CSMR.2013.13>
- [8] Emelie Engström and Per Runeson. 2011. Software product line testing - A systematic mapping study. *Information & Software Technology* 53, 1 (2011), 2–13.
- [9] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29–October 3, 2014*. IEEE Computer Society, 391–400. <https://doi.org/10.1109/ICMSE.2014.61>
- [10] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16–24, 2015, Volume 2*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 665–668. <https://doi.org/10.1109/ICSE.2015.218>
- [11] Charles W. Krueger. 2006. New methods in software product line practice. *Commun. ACM* 49, 12 (2006), 37–40. <https://doi.org/10.1145/1183236.1183262>
- [12] Jacob Krüger, Mustafa Al-Hajjaji, Sandro Schulze, Gunter Saake, and Thomas Leich. 2018. Towards automated test refactoring for software product lines. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10–14, 2018*, Thorsten Berger, Paulo Borba, Goetz Botterweck, Tomi Männistö, David Benavides, Sarah Nadi, Timo Kehrer, Rick Rabiser, Christoph Elsner, and Mukelabai Mukelabai (Eds.). ACM, 143–148. <https://doi.org/10.1145/3233027.3233040>
- [13] Roberto Erick Lopez-Herrejon, Stefan Fischer, Rudolf Ramler, and Alexander Egyed. 2015. A first systematic mapping study on combinatorial interaction testing for software product lines. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13–17, 2015*. IEEE Computer Society, 1–10. <https://doi.org/10.1109/ICSTW.2015.7107435>
- [14] Mukelabai Mukelabai, Damir Nesic, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling combinatorial explosion: a study of industrial needs and practices for analyzing highly configurable systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3–7, 2018*, Mari-anne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 155–166. <https://doi.org/10.1145/3238147.3238201>
- [15] Rudolf Ramler and Werner Putschögl. 2013. Reusing Automated Regression Tests for Multiple Variants of a Software Product Line. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18–22, 2013*. IEEE Computer Society, 122–123. <https://doi.org/10.1109/ICSTW.2013.21>
- [16] David J. Sheskin. 2007. *Handbook of Parametric and Nonparametric Statistical Procedures* (4 ed.). Chapman & Hall/CRC.
- [17] Mats Skoglund and Per Runeson. 2004. A case study on regression test suite maintenance in system evolution. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings*. IEEE, 438–442.

# Mutation Operators for Feature-Oriented Software Product Lines

Jacob Krüger

Otto-von-Guericke University  
Magdeburg, Germany  
[jkrueger@ovgu.de](mailto:jkrueger@ovgu.de)

Thomas Leich

Harz University & METOP GmbH  
Wernigerode & Magdeburg, Germany  
[tleich@hs-harz.de](mailto:tleich@hs-harz.de)

Mustafa Al-Hajjaji

pure-systems GmbH  
Magdeburg, Germany  
[mustafa.alhajjaji@pure-systems.com](mailto:mustafa.alhajjaji@pure-systems.com)

Gunter Saake

Otto-von-Guericke University  
Magdeburg, Germany  
[saaake@ovgu.de](mailto:saaake@ovgu.de)

## ABSTRACT

In this extended abstract, we describe the *Journal First* summary of our article with the same title published in the Journal of Software: Testing, Verification and Reliability (STVR) [1].

## CCS CONCEPTS

- Software and its engineering → Software product lines; Software testing and debugging.

## KEYWORDS

Mutation testing, software product line, variability faults

### ACM Reference Format:

Jacob Krüger, Mustafa Al-Hajjaji, Thomas Leich, and Gunter Saake. 2019. Mutation Operators for Feature-Oriented Software Product Lines. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342372>

Software product lines allow to systematically reuse software artifacts to implement a set of similar variants from the same codebase. The software artifacts are assigned to features that describe a specific functionality and can be enabled or disabled to customize each variant. This variability introduces additional complexity, increasing the costs to test a software product line to ensure the correct behavior of derived variants. Consequently, several testing and sampling techniques have been adapted to improve the efficiency and effectiveness of software-product-line testing.

A particularly challenging technique is mutation testing, where faults are automatically injected into a system to assess the quality of the test suite (not the system itself). The test suite is considered to be appropriate if it kills (its test cases fail) the generated mutants. However, mutation testing itself is an expensive technique, as a large number of mutants can be generated, and each must be run against all tests. For software product lines, this problem becomes

even more drastic, as the number of possible variants that must be mutated and tested can exponentially increase with each feature.

Researchers have proposed several approaches to tackle this explosion of costs, mainly cost reduction techniques for mutation testing and sampling variants of a software product line. In our article [1], we propose a set of seven mutation operators for feature-oriented programming that we designed to inject variability faults. Precisely, our operators mutate the mapping between feature model and artifacts as well as variability in the source code. Our idea is to reduce the costs of testing features and their interactions by limiting the number of injected mutations and focusing on actual variability faults. As our operators mutate variability in a software product line, we refer to such operators as *variability-aware mutation operators*. We derived most of these operators from existing ones (e.g., for preprocessors) by adopting them for feature-oriented programming.

To evaluate our mutation operators, we conducted an extensive empirical analysis, including four software product lines as subject systems and comparing our variability-aware with conventional (not specifically designed for variability) mutation operators. The results indicate that our proposed operators are indeed injecting variability faults more controlled than conventional operators. However, we also find that our operators can lead to variants that we cannot compile and that the number of redundant and equivalent mutants is problematic. Consequently, improved cost reduction techniques are necessary to address mutation testing of software product lines. Besides this experience, we also learned that the usability of variability-aware operators heavily depends on how developers employ the variability mechanism (i.e., feature-oriented programming), that automation for such operators is needed, and that test cases must consider the whole software product line. Namely, test cases must be variable by themselves to allow automated mutation testing and they have to test variability to kill the injected mutants. We faced problems with these two issues, as the tests of our subject systems did only partly test variability and were sometimes not usable for all variants.

**Acknowledgments.** This research is supported by the German Research Foundation (LE 3382/2-1, LE 3382/2-3, SA 465/49-1, SA 465/49-3) and Volkswagen Financial Services AG.

## REFERENCES

- [1] Jacob Krüger, Mustafa Al-Hajjaji, Thomas Leich, and Gunter Saake. 2019. Mutation Operators for Feature-Oriented Software Product Lines. *Software Testing, Verification and Reliability* 29, 1-2 (2019), e1676. <https://doi.org/10.1002/stvr.1676>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7138-4/19/09.

<https://doi.org/10.1145/3336294.3342372>

# Extended Abstract of “Spectrum-Based Fault Localization in Software Product Lines”

Aitor Arrieta

Mondragon University  
arrieta@mondragon.edu

Sergio Segura

University of Seville  
sergiosegura@us.es

Urtzi Markiegi

Mondragon University  
umarkiegi@mondragon.edu

Goiuria Sagardui

Mondragon University  
gsagardui@mondragon.edu

Leire Etxeberria

Mondragon University  
letxeberria@mondragon.edu

## ABSTRACT

Testing Software Product Lines (SPLs) is a challenging approach due to the huge number of products under test. Most of the SPL testing approaches have proposed novel ideas to make verification and validation activities cost-effective. However, after executing tests and detecting faults, debugging is a cumbersome and time consuming task. In our article [1], we proposed a debugging approach to localize bugs in SPLs that works in two steps: first, feature sets of the SPL are ranked according to their suspiciousness (i.e., likelihood of being faulty) by applying spectrum-based fault localization techniques. In a second step, a fault isolation algorithm is used to generate valid products of minimum size containing the most suspicious features, helping to isolate the cause of failures.

## CCS CONCEPTS

• Software and its engineering → Software product lines.

## KEYWORDS

Debugging; Software Product Lines; Spectrum-Based Fault Localization

### ACM Reference Format:

Aitor Arrieta, Sergio Segura, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. 2019. Extended Abstract of “Spectrum-Based Fault Localization in Software Product Lines”. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342369>

## 1 EXTENDED ABSTRACT

Testing Software Product Lines (SPLs) is a tedious and time consuming activity. Debugging SPLs, which is part of the testing process, is challenging due to the difficulty to find and isolate the faulty features in the SPL. In [1], we proposed an approach to SPL debugging, which works in two steps. First, the outcomes of testing are used to rank feature sets based on their suspiciousness score, which is a metric that measures the probability of a feature set having faults. This is performed by adapting the application of Spectrum-Based

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7138-4/19/09.

<https://doi.org/10.1145/3336294.3342369>

Fault Localization (SBFL) at the feature level, rather than at the code level, as conventional SBFL approaches [2]. Then, a fault isolation approach is proposed to generate products of minimum size containing the most suspicious features. This is done by automatically analyzing the feature model. Having products of minimum size facilitates engineers the isolation of the failure causes.

We empirically evaluated our approach by comparing the effectiveness of ten state-of-the-art SBFL techniques on nine SPLs of different sizes with seeded faults and combinatorial product suites. The results show that:

- By using the appropriate SBFL techniques, faults could be localized by examining between 0.1% and 14.4% of the feature sets.
- Three techniques stood out over the rest: Kulcynski2, Tarantula and Ample2.
- The accuracy of the approach was better when the number of products in the suite increased. The reason for this is that when the number of products in the suite is higher more information is available to compute the suspiciousness of the feature sets.
- The number and type of faults had a strong impact in the effectiveness of the suspiciousness techniques. As expected, isolating single faults was easier than locating multiple faults. In addition, locating faults caused by the interaction among different features was harder than locating faults caused by a single feature.

The advances on SPL testing contrast with the low number of approaches that support SPL debugging activities. Overall, the approach we proposed was effective to localize typical SPL bugs. This work complements the different cost-effective techniques proposed by the scientific community on testing SPLs and paves the way for new contributions on debugging SPLs.

## REFERENCES

- [1] Aitor Arrieta, Sergio Segura, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. Spectrum-based fault localization in software product lines. *Information and Software Technology*, 100:18–31, 2018.
- [2] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

# Applying Product Line Testing for the Electric Drive System

Rolf Ebert\*

Development Electric Drive, BMW  
Group, Munich, Germany  
[www.bmw.de](http://www.bmw.de)

Matthias Markthaler

Development Electric Drive, BMW  
Group, Munich, Germany  
RWTH Aachen University, Software  
Engineering, Aachen, Germany  
[www.bmw.de](http://www.bmw.de)

Jahir Jolianis

Development Electric Drive, BMW  
Group, Munich, Germany  
[www.bmw.de](http://www.bmw.de)

Benjamin Pruenster

Development Electric Drive, BMW  
Group, Munich, Germany  
[www.bmw.de](http://www.bmw.de)

Stefan Kriebel

Development Electric Drive, BMW  
Group, Munich, Germany  
[www.bmw.de](http://www.bmw.de)

Bernhard Rümpe

RWTH Aachen University, Software  
Engineering, Aachen, Germany  
[www.se-rwth.de](http://www.se-rwth.de)

Karin Samira Salman

Development Electric Drive, BMW  
Group, Munich, Germany  
[www.bmw.de](http://www.bmw.de)

## ABSTRACT

The growth in electrification and digitalization of vehicles leads to increasing variability and complexity of automotive systems. This poses new challenges for verification and validation, identified in a Product Line Engineering case study for the electric drive system. To overcome those challenges we developed a Product Line Testing methodology called TIGRE. In this paper, we present the TIGRE methodology. TIGRE comprises the identification and documentation of relevant data for efficient product line testing and the application of this data in the test management of an agile project environment. Furthermore, we present our experiences from the introduction into a large-scale industrial context. Based on our results from the introduction, we conclude that the TIGRE approach reduces the testing effort for automotive product lines significantly and, furthermore, allows us to transfer the results to untested products.

## CCS CONCEPTS

• Computer systems organization → Embedded systems; Redundancy; Robotics; • Networks → Network reliability.

## KEYWORDS

Product Line Engineering, Product Line Testing, Software Product Lines, Automotive Industry

\*The authors list is in alphabetical order.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336318>

## ACM Reference Format:

Rolf Ebert, Jahir Jolianis, Stefan Kriebel, Matthias Markthaler, Benjamin Pruenster, Bernhard Rümpe, and Karin Samira Salman. 2019. Applying Product Line Testing for the Electric Drive System. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3336294.3336318>

## 1 MOTIVATION

The automotive industry is facing several challenges ranging from the growing complexity of systems and software [13] to a customers demand for a shorter time-to-market [34]. To overcome those "automated, connected, electrified and service" challenges the BMW Group set up a strategy [3]. One important part of this strategy is the electrification of the drive system. In the past, the development of the electric drive system focused on individual derivates, e.g., the i3 and the i8. This focus changed and by the year 2025, a broad portfolio of approx. 25 derivates will be electrified [2].

The combination of variants of hardware, software, functionalities, time deviations, and other characteristics exponentially increases the scope to be tested. The combination possibilities enable the BMW Group to offer more customized products. At the same time, it challenges the validation and verification of the products. Approaches to deal with these challenges are Product Line Engineering (PLE) and Software PLE, respectively [8].

There is a lack of Software PLE experience reports from the industry [29]. The few industrial Software Product Line Testing (PLT) approaches focus on solving narrow research challenges instead of the whole product line process [28]. Hence, there is a lack of evidence about the transformation of an application-oriented Verification and Validation (V&V) to PLT in an industrial context.

In a case study for the electric drive system we identified industrial V&V challenges and compared them to existing PLE concepts. Based on the investigation's results, we developed a methodology, called Testing IntelliGent, non-Redundant and Efficient (TIGRE). Hence, our contributions are

- an approach to introduce PLT-methodologies into agile development processes
- TIGRE, a methodology for efficient testing of an automotive product line

The particular goal of TIGRE is to reduce redundant testing effort by intelligent planning and reporting of the test execution.

In the following, Section 2 introduces the electric drive system, before Section 3 presents the challenges related to V&V of this system in the industrial context. Section 4 presents TIGRE and Section 5 presents the results of our case study and lessons learned. Afterwards, Section 6 highlights related work. Section 7 discusses TIGRE and Section 8 concludes.

## 2 THE ELECTRIC DRIVE SYSTEM

To electrify a wide range of vehicles until 2025, the BMW Group developed the fifth generation of the electric drive system [2]. This fifth generation shows the typical characteristics of a product line. The following section introduces the most relevant characteristics.

### 2.1 Hardware-Components and Communication Busses

The fifth generation of BMW Group's electric drive systems comprises several hardware components. These components communicate over different bus systems. Most prominent are the highly integrated electric drive component, including the electric machine, transmission and power electronics, the battery system and the charger unit with Alternating Current (AC), Direct Current (DC) and DC/DC converter [2]. Each component is available in different variants. Hence, the system can be modified to suit all sorts of different installation spaces and power requirements.

### 2.2 Software-Functions and Parameters

The current electric drive system provides more than 60 software intensive functionalities, which are distributed over various hardware components. This distribution may vary for different vehicle derivates<sup>1</sup>. Among the main functionalities, e.g., torque, and energy management, plug-in charging of the electric vehicle is one with the highest degree of variability. Reasons are, e.g., country-specific legal requirements, grid voltages, charging standards, derivate specific vehicle architectures and optional equipment. There are different means to realize this functional variability in the software: One possibility is to implement different software variants; more precisely separate pieces of software, which are implemented in parallel. Another possibility is to have only one piece of software but a flexible parameter-set for each functional variant [41].

A future challenge is that proprietary embedded systems become linked to other systems that are more flexible, interactive, networked and seamlessly connected to Cyber-Physical Systems [7]. One of the first functionalities affected by this trend is the charging functionality. The coming into effect of the ISO15118 [36] sets the foundation for Vehicle-to-Grid functions and wireless high-level communication between the vehicle and the charging station [27].

<sup>1</sup> In the automotive industry, the configuration of a vehicle in a vehicle family is referred to as a *derivate* or model. The BMW 7 Series vehicle family, for example, is configured as a limousine derivate, long version derivate, hydrogen drive derivate, and hybrid derivate.

Summing up, each derivate comprises multiple variants, which differ in hardware, software, and the respective functionalities.

### 2.3 Development Process

In previous generations, the development process of the electric drive system focused on single derivates. For each derivate there existed dedicated responsibilities and artifacts like project plans, requirements, architectures, and test specifications, e.g., test cases, test plans, and test reports. Hence, transforming the system into a product line means the development process has to be changed. The following two prerequisites have already been established and must be taken into account.

*Functional- and Requirements-Based testing.* Concurrently with the development process, the emerging functionalities are tested to their requirements. The Specification Method for Requirements, Design, and Test (SMArDT) supports the process [10, 11].

*Agile Sprints.* At the BMW Group, software and system development are based on agile methods, e.g., Scrum and Large-Scale Scrum [18, 37]. According to these agile methods, new hardware and software are developed and integrated into sprints. Each sprint contains a specification, a development, and a testing phase.

## 3 CHALLENGES IN THE ELECTRIC DRIVE SECTOR

With the fifth generation of the electric drive system, a powerful product line has been established. This product line is the basis for developing derivates.

To handle the product line and its derivates, processes and artifacts have to be introduced, since this cannot be handled efficiently with the traditional application-oriented methods. One suitable methodology field is PLE and Software PLE, respectively. This section presents the main challenges, which we identified in a case study with the support of domain experts at the BMW Group. These challenges need to be addressed by the PLE approach in the automotive industry.

### Establishing Domain and Application Engineering (CH01)

To deploy PLE strategies, it is crucial to establish a common understanding of the difference between domain and application engineering. In [33] these terms are defined as follows:

**Domain Engineering** is the process of PLE in which the commonality and the variability of the product line are defined and realized.

**Application Engineering** is the process of PLE, in which the applications of the product line are built by reusing domain artifacts and exploiting the product line variability. In the automotive context the applications are derivates or subsystems of derivates. In a subsystem, like the electric drive system, we also call applications products.

**Common Platform** The common platform contains all elements from which the products can be built.

Traditionally the development of the electric drive system at the BMW Group is application-oriented. This is because in the beginning, when demand for E-Mobility was not as high as today, the BMW Group focused on two vehicle types, the i3 and the i8. Hence, one of the main challenges in establishing domain engineering is to reveal and document the explicit elements of the common platform. This documentation has to include the element dependencies and variabilities. The information has to be extracted from the existing application-specific artifacts and expert knowledge. In addition, the existing processes and artifacts for applications have to be adapted.

# Using Synergies of the Product Line for V&V (CH02)

With the increasing number of variants and the combination of variants, the testing effort for the electric drive system expands exponentially. Since the products have a high degree of reuse of the common platform elements, testing each single variant combination is not necessary. There are synergies between the products. The challenge is to reveal them, and by doing so enable intelligent test planning, which leads to an efficient V&V of the electric drive system.

## V&V of Product Lines in an agile development process (CH03)

In an agile development process, each sprint needs a suitable test plan for integration testing. The test plan has to consider the necessary test runs for all testing activities during the sprint. The test plans must not be separate for each derivate. Furthermore, the test plan has to ensure that all variants of the derivates are sufficiently tested and there are no gaps or redundancies. The test runs of the test plan must be distributed among the available test platforms in a way that no conflicts arise, e.g., two test runs on the same test platform at the same time. The challenge is to include all necessary information in the test plan, while being prepared for unforeseen events.

**Various Testing Objectives.** The purpose of testing may differ in each sprint. Depending on the changes in hardware, software and the status of the project, different test goals will be focused, e.g.,

- Verification of new functionality
  - Verification of new maturity levels with more robustness
  - Verification of bug fixes
  - Regression testing of unchanged functionality
  - Focus on a special variant or derivate, e.g., if it is close to start-of-production or before a specific milestone like homologation of new components, variants or derivates

In each agile sprint, there is a limited testing time. Since the number of variants increases exponentially, it is not feasible to increase the test resources in the same way. Hence, it might not always be possible to execute all available test cases for each variant in every cycle. It is, therefore, necessary to exploit synergies between the products to efficiently use existing resources.

*Flexibility of the Test Plan.* Even if a test run is perfectly planned, unplanned obstacles can occur during a sprint, e.g., delayed delivery or damage of software, components or test platforms. To use the

remaining testing time efficiently, short-term changes in the test plan must be possible at any time.

## Reporting the Test Results (CH04)

Another challenge is reporting test results to the common platform and derivate management. Besides the high expectations regarding quality, the electric drive system needs to ensure safety. The management and domain experts demand specific reports to the results of the test execution. These reports have to be relevant to derivate-focused management and to the common platform management. In contrast to the other challenges, CH04 is focused on an automotive specific subject. Automotive-specific, since the product (electric drive system) is always integrated into a specific derivate and additional safety reporting is required for each derivate.

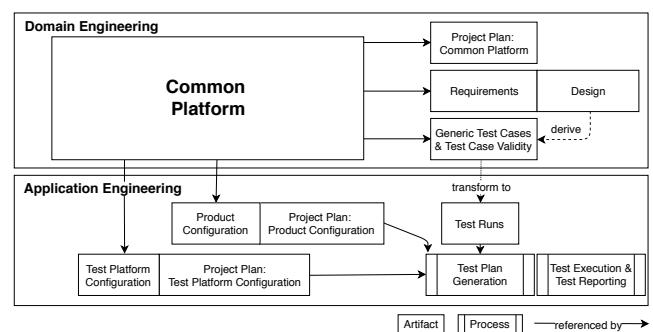
## 4 INTRODUCTION OF PLE METHODOLOGIES IN AUTOMOTIVE TESTING WITH THE TIGRE APPROACH

In this section, we propose the TIGRE approach to introduce PLE methodologies in the automotive industry and examined the testing process of the electric drive system. TIGRE targets the design of test plans for product lines in a non-redundant and efficient way. Hence, TIGRE targets the challenges of Section 3.

- 4.1 Phase one** aims at the initial identification and documentation of relevant data for PLT.
  - 4.2 Phase two** focuses on the usage of this data for testing activities during each agile sprint.

#### 4.1 Identification of the relevant data for PLT

The first phase of TIGRE addresses the establishment of Domain and Application Engineering (*cf.* **CH01**). This phase needs a high initial effort. Yet, once it is established the activities only have to be repeated in case of changes. Figure 1 gives an overview of the necessary artifacts.



**Figure 1: Artifacts and their relations in domain and application engineering**

The process of domain engineering comprises the common platform and the corresponding project plan, requirements, design, generic test cases, and test case validity. The process of application engineering comprises the product configurations and test platforms with their respective project plans, and test runs. All those

application engineering artifacts are used for test plan generation in the second phase (*cf.* Section 4.2). The following sections explain each artifact in detail.

**4.1.1 The Common Platform.** Defining the common platform is the key element for all further activities. The common platform contains all elements from which the products can be built. Each element is described as a variation point according to the Orthogonal Variability Model (OVM) [33]. There are several variants for each variation point. The defined elements of the common platform will subsequently be referred to in all further artifacts, like requirements, design and test specifications, project plans, and for the configuration of the products (see Figure 1).

The high degree of variability in hardware, software, and functionality of the electric drive system results in a large number of variation points. These variation points can be clustered into several groups, which we named *sub-platforms*. As a result, we defined the following sub-platforms:

- A platform of distributed functionalities
- A platform of hardware components
- A platform of software and parameters
- A platform of further characteristics, which give extra information about the electric drive system

Figure 2 illustrates the sub-platforms, including examples for their content. The sub-platform *Functionality* contains one variation point (triangle in Figure 2) for charging functionality (CHGE). CHGE has the variants (square) charging with AC, charging with DC and inductive charging (IND). The sub-platform *Hardware* comprises the variation points high voltage battery (HV-Battery), with the variants A and B, the electric motor, transmission and power electronics (abbreviation Power Electronics and Electric Machine (PEEM)), with the variants small S, medium M and large L, and the charging unit (CU), with two variants for high power charging (High) and low power charging (Low). The sub-platform *Software* contains the variation points Charging Unit Software (CU-SW) and the PEEM Software (PEEM-SW). Each variation point has two variants, *i.e.*, SW-A and SW-B. The sub-platform *Characteristics* comprises one variation point, the charging type (CHGE Type). The CHGE Type represents four different charging standards. The charging standard Type-1 is an American standard for AC-charging [19], Type-2 is a European standard for AC-charging [21, 22], Combined Charging System (CCS) is an international standard for combined AC-charging and DC-charging [23], and CHDMO is a Japanese standard for DC-charging [20].

**4.1.2 Dependencies within the Common Platform.** As described in [33] there exist certain dependencies between the elements of the common platform. These dependencies need to be analyzed and documented in order to show valid combinations. Figure 2 illustrates an example of such dependencies: The charging types Type-1, Type-2, and CCS *require* the functionality AC. The AC functionality *requires* a CU and an HV-Battery in any variant. The CU *requires* a CU-SW in any of the available variants.

**4.1.3 Requirements, Design and Test Specification.** The defined elements of the common platform shall be referred to in the requirements, design and test specifications during domain engineering.

For the electric drive system, this is done according to SMArDT: Model-based requirements and design artifacts are elaborated for each functionality of the common platform. Moreover, the artifacts contain information about functional variability. An example of the functional variability is the different behavior of the AC-Charging functionality for the charging standards Type-1 and Type-2.

Based on the SMArDT methodology, generic test cases are generated for each functionality [10, 11]. A generic test case corresponds to a domain test case [33]. Hence, a domain test case can contain variability and should allow efficient reuse in application testing. The set of domain test cases has to cover all functional variants. To realize this, we either define different test cases for each variant or one test case with variable parameters. Since this methodology for test case creation has already been established and published [10, 11], we will not go further into detail. We assume the resulting artifacts are available to the testing activities.

**4.1.4 Test Case Validity.** The test case validity is an attribute of each generic test case which indicates for which objects the test case is valid and executable.

In the past, due to the application-oriented approach, the test case validity has been documented through an attribute listing for all relevant derivates. With the increasing variability, this is not applicable anymore, because:

- Each derivate may comprise different variants (product configurations) and a test case may not be valid for each variant
- The test case would be executed for each derivate once and there would not be any use of the synergies between the derivates

Therefore, TIGRE comprises a PLE concept of expressing the test case validity based on the defined elements of the common platform. This allows us to use the synergies of the product line as required in **CH02** and reduce testing effort. Instead of performing the test case for each derivate separately, we perform it depending on the common platform elements, which are stated in the validity expression. If the elements are reused by several derivates, the test result is valid for all those derivates. We want to emphasize that this holds for integration testing. Hence, not all components and sub-systems are integrated into the derivate yet. For feature interactions to be tested, this approach refers to the system tests that follow the integration test.

A generic test case can be valid for several variants of elements of the common platform. Hence, two different scenarios for the execution are possible:

- The test case must be performed with several runs, one for each variant it is valid for
- The test case must be performed with only one run for any variant it is valid for and the result can be transferred to the other variants it is valid for

The second scenario allows us to reduce testing effort again and therefore contributes to **CH02**. In both scenarios, each test run can be performed with a different parameterization, depending on the validity of the test case.

For the test case validity, we use information provided in the requirements and design specifications as well as domain experts assessments. The transparency, why a test case needs several runs, is

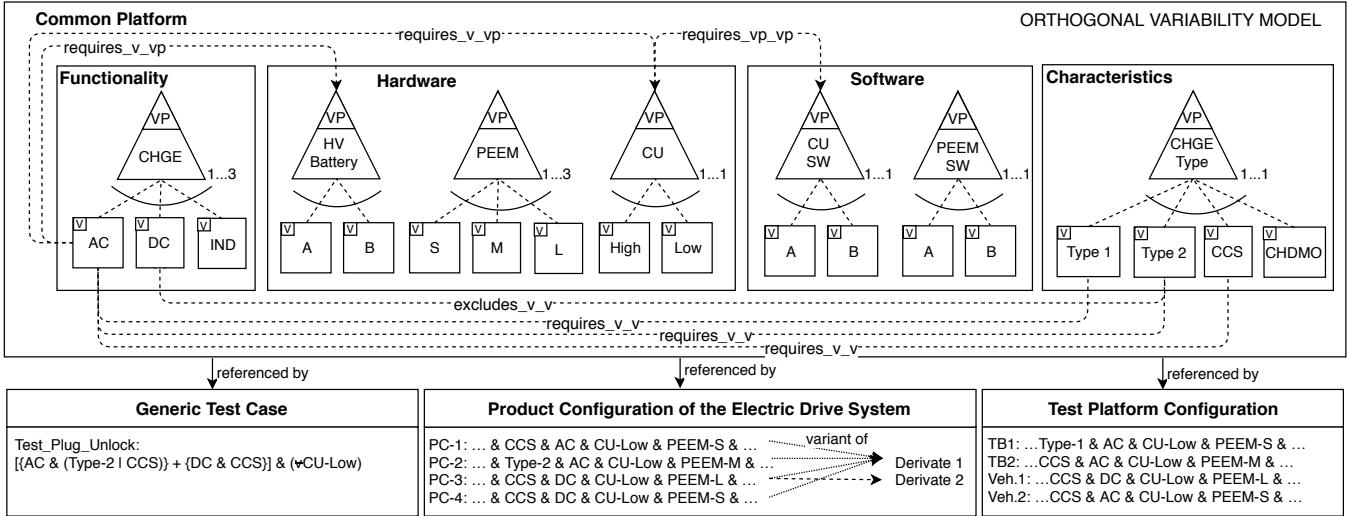


Figure 2: Example of artifacts of the common platform, generic test case, product configuration and test instance configurations

increased compared to a derivate relationship. The test case validity is stored as an attribute of the generic test case. We can change and improve it throughout the development process of the product line. This allows us to take advantage of lessons learned and document the expert knowledge in a reasonable way (*cf. CH01*).

To use the test case validity for the planning of the test execution and reporting, it is documented in a specific notation. To express the explained scenarios for test case validity, a notation with logical expressions is feasible. In addition to being machine-readable, this logical expression must also be readable by domain experts, test experts and managers. Therefore, the syntax should be minimalistic in order to facilitate readability and interpretation.

In particular, we identified the need to expand the test case validity expressions by equivalence classes. An equivalence class is a set of variants which a generic test case is valid for. This specific set states that we need only one run of the test with any representative of the equivalence class. The result of this single run can be transferred to all other members of the equivalence class.

In the following, we explain the Operator-Name, the Operator-Symbol, the Operator-Description and the Operator-Example used in our specific notation.

- **Symbol:** |  
**Name:** logical OR  
**Example:** A | B

**Description:** The test case is valid for A and B, it must be executed once for a product containing either A or B and the result can be transferred to all other products containing A or B. This operator is used to express equivalence classes.

- **Symbol:** &  
**Name:** logical AND  
**Example:** A & B

**Description:** The test case is valid for A and B, it must be executed once with a product containing A and B.

- **Symbol:** {...}+{...}  
**Name:** Separate-Runs-Operator

#### Example: {A} + {B}

**Description:** The test case is valid for A and B, it must be executed twice, once with a product containing A and once with a product containing B.

- **Symbol:** ∨

**Name:** All-Operator

**Example:** ∨Variation-Point-X

**Description:** The test case is valid for all variants of Variation-Point-X, it must be executed separately for each variant.

As an example, we use a generic test case concerning the unlock functionality of the charging plug. The following expression shows its validity attribute (*cf. Figure 2*) i.e., Test\_Plug\_Unlock:

$\{(\text{AC} \& (\text{Type-2} \mid \text{CCS})) + (\text{DC} \& \text{CCS})\} \& (\forall \text{CU-Low})$

With this validity attribute, we conclude that we need two separate runs of the test case Test\_Plug\_Unlock. One test run with a product containing the variants AC, and either Type-2 or CCS and another run with a product containing the variants DC and CCS. The expression  $(\text{Type-2} \mid \text{CCS})$  forms an equivalence class. Both runs have to be executed with a CU-Low variant. The all-operator states that all versions of CU-Low have to be tested separately. This will be explained in detail in Section 4.1.7.

**4.1.5 Product Configurations.** The specific products are managed within the application engineering. A major principle of PLE is that the products are composed of a combination of the reusable elements of the common platform. The specification of the variation points, including their dependencies, enables the generation for possible product configurations.

To set up a complete product configuration, we consider all variation points of the common platform and decide whether they are present or not. If a variation point is present we select at least one of the possible variants. This is in accordance with the defined dependencies and restrictions, as described in Section 4.1.2.

The resulting product configurations are stored in a database. Each configuration is mapped to the derivate where it is present. Figure 2 shows four examples for parts of test-relevant Product Configurations (PC) and their mapping to product, e.g.,

$PC-2 : \dots \& Type-2 \& AC \& CU-Low \& PEEM-M \& \dots$

In the context of product configuration (*cf. PC-2*), it is a list of the elements that make up the product. Similar to the test platform configuration the elements are connected with the symbol  $\&$ .

**4.1.6 Test Platform Configurations.** Similar to the product configurations, the test platform configurations are expressed by the common platform elements. A test platform is a test bench or vehicle where a test run can be executed. This expression of the test platform configuration allows easy matching of test case validity, product configurations, and test platforms when planning the test execution. Examples for test platform configurations are depicted in Figure 2. Test Bench 1 has the following configuration:

$TB1 : \dots \& Type-1 \& AC \& CU-Low \& PEEM-S \& \dots$

**4.1.7 Project Plans.** As a prerequisite for planning the test execution for agile sprints, not only *variability in space* but also *variability in time* has to be considered. The variability in time is the existence of different versions of an element of the common platform that are valid at different times [33]. The variability in space is the existence of an element in different shapes at the same time. Therefore, for each variant of the common platform, we have to define whether it is available and in which version, for each specific sprint. An example is depicted in Figure 3. The example shows the different versions of the hardware component variants CU-Low and PEEM-S during four successive sprints. Moreover, it illustrates, in which sprints the functionality AC is available. In sprint 1, both component variants are available in version 1 (v1), but AC is not present. In sprint 2, CU-Low is available in v2, while PEEM-S is still available with v1 only. In this sprint, the AC-functionality is present for the first time. From now on it will be available in all further sprints. In sprint 3, CU-Low is available in v3 and PEEM-S in v2. In sprint 4 CU-Low is available in v4 and PEEM-S in v3. We call this the *project plan of the common platform*.

Project Plan of the Common Platform							
Sprint 1	◇	Sprint 2	◇	Sprint 3	◇	Sprint 4	◇
CU-Low							
v1	◇	v2	◇	v3	◇	v4	◇
PEEM-S		v1		v2		v3	◇
AC							
NO	◇	YES	◇	YES	◇	YES	◇

Project Plan of the Products			
Product 1 (Derivate 1)	CU-Low: V1	CU-Low: V2	CU-Low: V3
CU-Low: V1			
PEEM-S: V1	PEEM-S: V1	PEEM-S: V1	PEEM-S: V2
AC: NO	AC: YES	AC: YES	AC: YES

Product 2 (Derivate 2)			
CU-Low: V1	CU-Low: V1, V2	CU-Low: V3	CU-Low: V4
CU-Low: V1			
PEEM-S: V1	PEEM-S: V1	PEEM-S: V1	PEEM-S: V3
AC: NO	AC: YES	AC: YES	AC: YES

**Figure 3: Simplified example of a sprint schedule**

To define, which version of a variant is integrated into a product for each agile sprint, a *project plan for products* is necessary as well.

Figure 3 illustrates, which versions of CU-Low and PEEM-S have to be tested for *product 1* and *product 2* during the four sprints. It also shows which products include the AC-functionality and in which sprint it will start to be present. Based on project decisions the following scenarios for integrating available versions from the common platform to products are possible:

- A newly available version in the common platform might be integrated at a later sprint for a special product. For example, in sprint 3 there is a v2 available for the PEEM-S. However, none of the products integrate it in this sprint.
- For one product, there might be two valid versions of the same variant at the same sprint. For example in sprint 2, product 1 uses both versions v1 and v2 of the CU-Low.

Thus, it is possible that different versions of one product configuration have to be tested separately during one sprint. As an example, we take the product configurations, which have been defined in the previous section (*cf. Figure 2*). If we want to test them in sprint 2, we have to consider the relevant versions defined for sprint 2 in the project plans (*cf. Figure 3*). This leads to the following *sprint-relevant product configurations*:

PC-1: CCS & AC & CU-Low-v1 & PEEM-S-v1 & ...  
 PC-2: Type-2 & AC & CU-Low-v1 & PEEM-M-v1 & ...  
 PC-3: CCS & DC & CU-Low-v1 & PEEM-L-v1 & ...  
 PC-4: CCS & DC & CU-Low-v2 & PEEM-L-v1 & ...  
 PC-5: CCS & DC & CU-Low-v1 & PEEM-S-v1 & ...

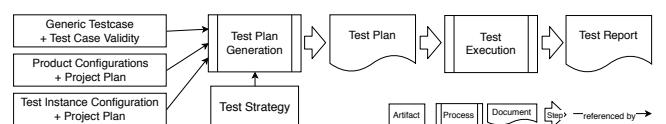
These sprint-relevant product configurations have to be considered during the planning of the test execution in the second phase of this approach.

There is also a project plan of the test platforms, which reveals the versions that will be available on the test platform at a certain sprint. From this project plan, the sprint-relevant test platform configurations can be determined with the same method.

## 4.2 Testing the Product Line in Agile Sprints

The second phase of TIGRE focuses on the planning of the test execution and the reporting of test results. It addresses the challenges **CH02**, **CH03**, and **CH04**.

Figure 4 gives an overview of this phase. In the beginning, relevant information of the before established artifacts is gathered. The data of each artifact are combined to generate a test plan for the test execution. The test plan has to comprise (1) the necessary runs of a generic test case, (2) the product configuration each run has to be performed with, and (3) the test platform each run has to be performed on.



**Figure 4: Overview of the interconnected artifacts for the test plan generation**

To optimize the test plan, different test strategies can be applied to the generation process. In each agile sprint, a suitable test strategy is selected to fulfill a specific testing objective. After the generation of the test plan, the tests are executed and the test results are reported. In the following sections, we provide a detailed explanation of the *test plan generation* and the *reporting of test results*.

#### 4.2.1 Generating the Test Plan.

The generation of the test plan can be divided into three steps.

**Step 1.** In step 1, the test runs that are relevant for a specific sprint have to be determined. To do this, we filter the project plan for all functionalities that are present in the specific sprint. Based on these functionalities, we choose the related test cases.

For each generic test case, we evaluate the validity attribute to identify which variants need separate runs of the test case. This results in a number of possible test runs. In this way, we transfer the generic domain test case to several test runs for application testing. This step corresponds to the binding of variability [28].

Figure 5 illustrates the calculated test runs for the validity attribute of the example generic test case presented in Figure 2. For the calculation, we also considered the available variant versions in sprint 2 (*cf.* Figure 3). Figure 5 depicts four separate test runs derived from the test case `Test_Plug_Unlock` (*cf.* Figure 2):

```
Run-1: AC & (Type-2 | CCS) & CU-Low-v1
Run-2: AC & (Type-2 | CCS) & CU-Low-v2
Run-3: DC & CCS & CU-Low-v1
Run-4: DC & CCS & CU-Low-v2
```

The first two test runs contain an equivalence class. To plan the test run for execution, we have to select one representative of the equivalence class. The decision for a suitable representative must be supported by the test strategy. Based on our analyses of frequent testing objectives during an agile sprint, the following test strategies meet our needs:

**Random:** For each equivalence class, we select a random representative.

**Exclusion** We define certain variants that should not be selected as a representative for an equivalence class.

**Preference** We define one or more specific variants that are preferably selected as a representative for an equivalence class.

**History-Based** For each equivalence class, we select a representative which has not been selected in the past.

**Maximal or Minimal Variation** For all equivalence classes, we select the representatives so that we receive the minimal or maximal possible number of different representatives.

**Combinatorial Interaction Testing (CIT)** CIT [12] enables the selection of a subset of products, which most likely reveal the faults caused by interaction. The representatives are selected so that relevant combinations of variants occur, but not all possible combinations. Hence, the testing effort can be reduced.

For our example, we select a history-based strategy and assume that CCS has been executed last time, so now we select Type-2. This means that the first test run will be executed for a product

containing AC, Type-2, and CU-Low-v1. The result will be transferred to all products, which include these variants and also to all products with CCS, AC, and CU-Low-v1.

As shown in Figure 5, each test run inherits a part of the logical validity expression from its parent generic test case. Since this validity expression consists of elements of the common platform, it also represents a part of a possible product configuration. Thus, we call the test runs validity expression a *sub-configuration*.

**Step 2.** In step 2, we extract the *sprint-relevant product configurations* as described in Section 4.1.7. In our example, the resulting configurations are depicted in Figure 5. Subsequently, we assign the sub-configurations of the test runs to the *sprint-relevant product configurations* (see (A) in Figure 5). This can lead to different results:

- If there is no product configuration containing a certain sub-configuration, the test run must be deleted. In our example, this is the case for the test run 2.
- If there is exactly one product configuration containing a certain sub-configuration, the test run must be executed with this configuration. This is the case for the test run 4.
- If there is more than one product configuration containing a certain sub-configuration, the test run can be performed on any product configuration. The test result can be transferred to the other product configurations. This refers to **CH02**. In the example, this holds for the test run 3.

In the last case, we must select one product configuration for the execution. To support a reasonable decision-making, the test strategies defined in step 1 can be used again.

In the example, we use a test strategy that selects the PEEM-L variant with *preference*. As a result, product configuration 3 will be assigned to test run 3.

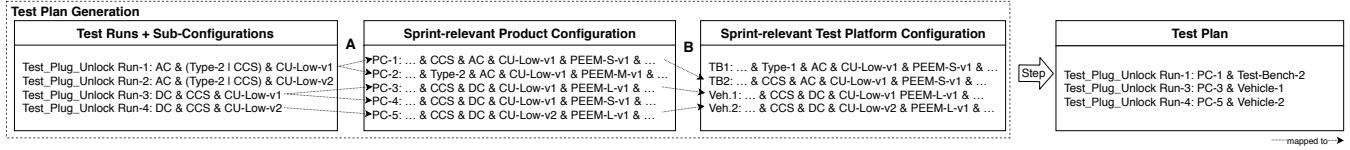
**Step 3.** In step 3, we determine the sprint relevant test platform configurations as described in section 4.1.7 and allocate the test runs to test platforms. To do this, we compare the selected product-configurations of each test run with the *sprint-relevant test platform configuration* (see (B) in Figure 5). If a test platform with the required configuration is available, the triple (test run – product configuration – test platform) can be assigned to the test plan. If no test platform is available, we consider the following alternatives:

- If the test run contains an equivalence class, we could take another representative
- If a test run's sub-configuration is present in several product configurations, we can select another product configuration
- If none of the above alternatives led to a result, we have to show the gap in the test report

For the first test run of the example in Figure 5, we recognize that there is no test platform with a sub-configuration AC & Type-2. Therefore, we select the other representative of the equivalence class and plan the run AC & CCS at test bench 2 (TB2).

Figure 5 shows the resulting test plan. It contains all test runs that need to be executed. For each test run, the product configuration and the test platform for the test execution are provided.

**4.2.2 Reporting the Test Results.** After the execution of the planned test runs, the test results are reported. Hence, we generate dedicated test reports for the common platform and the commercially



**Figure 5: Example of the test plan generation, for a single generic test case, with consideration of the test strategy**

relevant products and derivates, respectively (*cf. CH04*). These reports provide different views on the test results. Hence, different Stakeholders in the Company get a precise overview of the test results, edited/prepared for their specific view. Those views are automatically derived from the existing test results. The first report lists each test run result, showing the sub-configuration, it was executed with and the configurations where the result can be transferred. The product specific report lists only those test run results that are valid for a sub-configuration of this products. For each test run, we report whether it was executed with this product or transferred from the execution with another product.

## 5 CASE STUDY AND LESSONS LEARNED FROM EMBEDDING PLT INTO THE AUTOMOTIVE INDUSTRY

After defining the main process and input artefacts, the proposed approach was applied during a pilot phase. The aim of the pilot project was to show the applicability in an industrial environment. Furthermore, the pilot phase already involved members of various teams and helped to spread the basic knowledge about PLE fundamentals and the methodologies.

In the beginning, the relevant data for PLT was elaborated as described in Section 4.1 of the TIGRE approach. Since there was no specific tool support, the complexity for the pilot phase was kept as low as possible. For an integration phase of the electric drive system seven representative drive system functionalities were chosen. For each of those functionalities all the linked test cases were imported from application lifecycle management tools. Hence, we received 163 generic test cases in total. We chose three derivates for which the pilot phase should provide test results. Based on the defined elements of the common platform we derived the sprint relevant product configurations. Although the amount of variation points in the characteristics sub-platform was low, we identified 30 sprint relevant product configurations for the three derivates.

### 5.1 Pilot phase

The goal of the pilot phase was the manual creation of a test plan for the chosen integration phase. In order to generate the test plan, the following artifacts were created:

- The sprint-relevant product configurations, based on interviews with experts and the information contained in several databases.
- The generic test cases with their validity attributes.
- The sprint-relevant test platforms with their configurations and availability.

After collecting all of the necessary data, exemplary test sets were created following the proposed approach in section Section 4. The

result of the pilot phase has been a significant reduction of test runs, compared to just multiplying all of the test cases with the number of sprint-relevant product configurations.

Executing each test case once for each product configuration would result in the multiplication of 163 test cases with the 30 product configurations. However, not all of those 4.890 runs would be reasonable to be executed, since there exist synergies.

Applying the TIGRE methodology, we could reduce the number of necessary test runs to a total of 326 runs, which covers all of the sprint-relevant product configurations for the electric drive system. Furthermore, we were able to trace our results back to all variants and expose those which had not been explicitly tested.

### 5.2 Lessons learned from applying the methodology

Creating a test plan with a generally defined methodology aims to improve the traceability, reproducibility and optimization of the test plans over time. However, collecting all of the relevant information and combining them was a challenging task.

The generation of the three above mentioned artifacts required access to various tools and expert knowledge as well. The agile development requires agile tools for data maintenance. Project decisions obtained during meetings are traditionally documented in various independent tools and transferred to the actual database in a second step. In this transfer process information could get falsified or even lost. Moreover, specialized tools are often only accessible and understood by experts. Hence, we collaborated closely with those experts. The effort was worth it, since we stored the expert knowledge in tools. Now, all of the expertise can be reused easily and even if there is a personnel change.

Though others see a big challenge in the acceptance of PLE with its development structure [9], TIGRE was welcomed with interest by all involved domain experts, testing experts and managers.

Because of the high complexity of the electric drive system, manually generating a test plan was not feasible with reasonable effort. Hence, the implementation of such a methodology with modern tools is necessary.

## 6 RELATED WORK

In the past twenty years, several literature reviews have been conducted on PLT, analyzing a number of research approaches in this field. However, in the research field of PLE, the number of studies facing the scale testing of Product Lines in the industry is low [14, 28, 29].

The Software PLE investigation [29] reviewed 276 studies with the scope of identifying test strategies, which facilitate fault detection and reduce the effort for quality assurance. The study considered studies between 1998 and 2013. Only 6% refer to industrial practices and 8% are industrial studies. The investigation concludes that Software PLE studies can be categorized in "selection of products" to test and "actual test of products". However, only in the year 2013 there were studies found taking into consideration both processes. CIT (*cf.* Section 4) is also seen as a principle to reduce the testing effort and targets the challenges of **CH02** and **CH03**. The fundamental assumption of CIT is that most of the errors are caused by a single value or the interaction of two input values [12]. The study [29] identified CIT as the most common approach to reduce testing effort through the selection of products. Hence, TIGRE also comprises this optimization technique.

Apart from one study validated in an industrial context, all other approaches were evaluated in research groups. In the field of test case generation, test case selection and test inputs [29] could not find any available tools. Moreover, the literature review provided no information about studies concerning the test management, test plan generation or test reports.

In [14] a systematic mapping study on Software PLT is presented. Test organization, test processes, and test management are pointed out as relevant fields. The majority of the studies provide proposals for the established challenges in PLT. Nevertheless, only 17% of the studies present the usage and evaluation of the proposed methods. Hence, most of the proposed methodologies assume the existence of idealistic inputs and state clear requirements for the approaches, *i.e.*, complete variability models and properly linked information in the overall development process. In an industrial environment, those methodologies have to prove that they work under industrial circumstances. Those applicable test methodologies and strategies are hard to find [14].

Another survey [28] compared and analyzed the contributions of eight software product line specific approaches. The survey provides a framework of relevant artifacts for the overall Software PLT process. The framework enables categorizing and analyzing existing researches, as well as the identification of open research opportunities. The most relevant revealed research opportunities for our research are:

- Criteria for test case selection to reflect the decision on variability resolution and application-specific variability
- Coverage mechanism to determine test case creation, test selection, and execution approaches.
- Reusing or/and modifying test assets for application-specific testing

The framework neither details the artifacts nor explains the artifact's interaction for direct and profitable industry application. Agile methods are not considered. The conclusion of [28] is that most of the approaches focus on narrow Software PLT research challenges. Hence, the approaches do not provide a holistic software product line process from the beginning until completion. Furthermore, the survey stated that only marginal research on how to reduce redundant testing effort by PLT techniques exist. This reduction of redundant testing effort by PLT is one of the most important challenges we address in this paper.

Besides the aforementioned literature reviews, several PLT studies address parts of challenges mentioned in Section 3. The methodology and tooling described in [35] are based on feature models and focus on Software PLE. The extensive approach tackles the challenges **CH1**, **CH2**, and **CH4**. It distinguishes between the problem domain and the solution domain. The problem domain, which is represented in feature models, expresses similarities and variabilities. Hence, the stakeholders' requirements emerge from this domain. The solution domain, a hierarchical structure, consisting of elements of the architecture, satisfy those requirements. Thus, this hierarchical structure comprises the variation points and the connection of the solution and problem space. The language resulting from the approach expresses constraints, restrictions, and moreover calculations. Similar to our specific notation the language uses operators, *e.g.*, **REQUIRES**, **IMPLIES**, and **CONFLICTS**, in combination with logical operators, *e.g.*, **AND** and **OR**, but with a different interpretation, not in context for testing. The approach focuses on software, hence its testing framework is mainly designed for software tests. As such, we did not find applicable solutions for our specific agile test plan generation. Still, the tool provides a well-developed language and its infrastructure provides a large set of extensions and interfaces.

The study [15] introduces a product line application in the automotive industry and presents approaches to meet the challenges of the "mega-scale" product line. The challenges **CH01** and **CH03** are also identified and necessary measures are presented. Mentioned challenges are product complexity, a high number of variants in automotive applications, feature interactions, the size of the companies, the traceability, and the consistency throughout the life cycle. We have faced those challenges in our study as well. To provide solutions to the architectural decomposition, involved roles in the development and the necessary organizational adoptions are discussed. Despite an overview of the PLE development process, the study does not cover details concerning the test process *i.e.*, test management and test reporting. The subsequent study [41] enhances the predefined "mega-scale" product line challenges with challenges addressing **CH02**. Furthermore, the approach takes advantage of system modeling and feature models. However, no applicable solutions for Software PLT, *i.e.*, the test plan generation and reporting of test results (**CH04**) are presented.

In [17] the challenges of testing the weapon system using PLE approaches and a commercial PLE tool are presented. The study refers to **CH01**, **CH02**, and **CH03**. In order to start testing activities so-called 'branching' is necessary. Thus, a copy of the actual work is done and this release is used as a stable basis for testing activities. If necessary changes to the copy must be done, a strong discipline regarding the changes on the copies is required. Event branches are also created, as during the development of the product line also products must be generated at specific points in time. Although the approach tackles testing under time pressure, the variation of the environment under consideration is comparatively low to the automotive industry. Hence, the presented "copy-approach" is not sufficient for the complexity of the electric drive system. Furthermore, besides capturing the results in a database, the approach does not detail the reporting of the results.

Another approach [6] focuses on **CH01** and presents the possibilities of modeling the problem and solution space with tool support.

The approach shows the design of the variable architecture and product derivation. Testing the product line, however, is mentioned as an open issue of Software PLE. The reduction of the testing effort is addressed as an issue concerning many product lines.

The approach in [30] targets the challenges of **CH02** by reducing the testing time of cyber-physical-systems. In the first step, the approach reduces the testing time by selecting and prioritizing products to be tested. In the next step, multiple testing iterations are performed. In each iteration, a small number of test cases for each product is allocated and the test results feedback is taken into account for the next products and test cases. The approach focuses on reducing testing time, while test platforms and information about the product project management are not considered. The test platforms and the knowledge about availability and deadlines for each drive system variant are essential for our approach. Furthermore, [30] does not consider the aspect of an agile development process (**CH03**) or reporting test results (**CH04**).

The model-based approach in [38] aims to avoid redundant testing in application engineering (**CH02**). In order to avoid the execution of redundant test cases, the approach introduces a dataflow-based coverage criterion. The approach combines the usage of feature models and activity diagrams as test models to link data accordingly. Dependencies between variants are registered using a dependency matrix. The conclusion of [38] is that there is a high potential for increasing test efficiency. However, there is a strong dependency on the implementation of the test cases. Test cases can cover more branches and more than one dependency. Thus this dependency influences the efficiency of the approach. In the context of the electric drive system, the technique of [38] is very interesting for the generation of test cases from system models. However, this only covers functional variability. Variability of hardware, software and time are not considered.

Another approach [32] uses pairwise testing to avoid testing every possible combination of input values (**CH02**). The approach only tests the combination of all pairs. Furthermore, this testing technique could be transferred from input values to the features in a product line. The approach uses CIT based on a feature model of the domain. Based on the dependencies of the features the approach can achieve a reduction of the test cases. Nevertheless, the approach does not address our challenges regarding agile process development and reporting test results.

## 7 DISCUSSION

Our experiences and conclusions are based on an industrial pilot project. To introduce TIGRE in the electric drive system, we carried out the work to a large extent manually. Although we only considered the electric drive system, we still consider the results to be meaningful because our approach involves relevant domains for many automotive functionalities, *i.e.*, mechanical, electrical, and software domain.

We are aware that feature interactions can be a risk of transferring test results [1, 4, 25, 26, 39, 40]. Nevertheless, it will not always be possible to perform every test with every product configuration, because of restricted time and test resources [39]. Therefore, we have to select the test runs carefully. By documenting the test case validity, TIGRE supports a transparent and reproducible selection

of test runs. Moreover, due to the high safety requirements in the automotive domain, possible interactions must be identified before implementation [24]. Hence, in contrast to the telecommunication domain, fewer errors are found due to feature tests that were not already found by functional tests [16]. For feature interactions we recommend methods in development and specific test cases that TIGRE can integrate in system testing [1, 5, 25, 31].

We consider CIT for TIGRE, but because of time constraints, we could not apply the method for the pilot phase. CIT could reduce our testing effort even further.

In the pilot phase, we derived the information about the validity expressions manually out of the SMarDT models and by expert assessments. Despite the high manual effort and the uncertainty about the correctness of the expert knowledge, the approach lead to a higher transparency for test case selection. The decisions can be replicated, discussed and changed by lessons learned.

All the information of existing derivates is stored in a database. Hence, we did not set up feature models or OVMs for the pilot phase. Although OVMs and feature models also contain non-existing configurations, we only need real configurations for our test runs. Our specific notation is simple and exactly adapted to the needs of testing the electric drive system. Due to our specific notation, we could not use an elaborated existing tool for our approach. Depending on the type of tool we use, we may transform our notation. Hence, our generated test plan with its runs only exists in one table form and has to be transferred to the test management tool in a second step. The generation of an executable test plan to the defined test strategy still has to be implemented.

## 8 CONCLUSION

The increasing number of electrified derivates over the next few years is urging the BMW Group to develop and apply new methodologies. Since the new generation of the electric drive system shows significant product line characteristics, one approach to maintain high-quality standards is the use of PLE methodologies.

TIGRE introduces PLE methodologies to the integration testing of the electric drive system. It comprises several artifacts that need to be established in domain and application engineering. These artifacts facilitate the planning of an efficient test execution with reduced test effort. Thus, we demonstrate a procedure to optimize the assignment of domain test cases, product configurations and test platforms.

In a case study with seven representative functionalities of the electric drive system, we reduced the redundant testing effort from 4.890 test runs to 326 test runs for one agile sprint phase. Moreover, we are able to transfer the test results from tested product configurations to other non-tested product configurations. Hence, the TIGRE methodology allows to take advantage of product line synergies and reduces test effort. It is capable of increasing the efficiency, traceability and transparency for the integration test of the electric drive system.

## ACKNOWLEDGMENTS

We thank all participating experts who have supported our work, shared their knowledge and evaluated the TIGRE methodology.

## REFERENCES

- [1] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2018. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Marianne Huchard (Ed.). ACM, [S.l.], 143–154.
- [2] Matzner Angelika. 28.12.2018. Interview: Stefan Juraschek, Vice President Development Electric-Powertrain. <https://www.press.bmwgroup.com/global/article/detail/T0288899EN/interview:-stefan-juraschek-vice-president-development-electric-powertrain?language=en>
- [3] Simon Anika. 07.09.2017. Statement Harald Krüger, Chairman of the Board of Management of BMW AG, IAA Preview 2017: Press Release. <https://www.press.bmwgroup.com/global/article/>
- [4] Sven Apel, Joanne M. Atlee, Luciano Baresi, and Pamela Zave. 2014. Feature Interactions: The Next Generation (Dagstuhl Seminar 14281). *Dagstuhl Seminar 14281* (2014).
- [5] Sebastian Benz. 2010. *Generating Tests for Feature Interaction*. Dissertation. Technische Universität München, München.
- [6] Danilo Beuche and Mark Dalgarno. 2007. Software product line engineering with feature models. *Overload Journal* 78 (2007), 5–8.
- [7] Manfred Broy, Maria Victoria Cengarle, and Eva Geisberger. 2012. Cyber-Physical Systems: Imminent Challenges. In *Large-scale complex IT systems*, Radu Calinescu and David Garlan (Eds.). Lecture Notes in Computer Science, Vol. 7539. Springer, Berlin, 1–28.
- [8] Paul Clements and Linda Northrop. 2002. *Software product lines: practices and patterns*. Vol. 3. Addison-Wesley Reading.
- [9] Paul C Clements. 2015. Product Line Engineering Comes to the Industrial Mainstream. In *INCOSE International Symposium*, Vol. 25. Wiley Online Library, 1305–1319.
- [10] Imke Drave, Timo Greifenberg, Steffen Hillermacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. 2019. SMArDT modeling for automotive software testing. *Software: Practice and Experience* 0, 49 (2019), 301–328.
- [11] Imke Drave, Timo Greifenberg, Steffen Hillermacher, Stefan Kriebel, Matthias Markthaler, Bernhard Rumpe, and Andreas Wortmann. 2018. Model-Based Testing of Software-Based System Functions. In *44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 146–153.
- [12] Irwin S Dunietz, Willa K Ehrlich, BD Szablak, Colin L Mallows, and Anthony Iannino. 1997. Applying design of experiments to software testing: experience report. In *Proceedings of the 19th international conference on Software engineering*. ACM, 205–215.
- [13] Christof Ebert and John Favaro. 2017. Automotive Software. *IEEE Software* 34, 3 (2017), 33–39.
- [14] Emelie Engström and Per Runeson. 2011. Software product line testing – A systematic mapping study. *Information and Software Technology* 53, 1 (2011), 2–13.
- [15] Rick Flores, Charles Krueger, and Paul Clements. 2012. Mega-scale product line engineering at general motors. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*. ACM, 259–268.
- [16] Brady J. Garvin and Myra B. Cohen (Eds.). 2011. *Feature Interaction Faults Revisited: An Exploratory Study: 2011 IEEE 22nd International Symposium on Software Reliability Engineering*.
- [17] Susan P. Gregg, Denise M. Albert, and Paul Clements. 2017. Product Line Engineering on the Right Side of the V. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*. ACM, 165–174.
- [18] Mathias Heerwagen. 2018. Entwicklung im Wandel Agile Methoden auf dem Vormarsch. *ATZ - Automobiltechnische Zeitschrift* 120, 4 (2018), 10–15.
- [19] Hybrid - EV Committee. 2017. SAE Electric Vehicle and Plug in Hybrid Electric Vehicle Conductive Charge Coupler: SAE J1772.
- [20] IEEE. 2016. Std 2030.1.1-2015: IEEE Standard Technical Specifications of a DC Quick Charger for Use with Electric Vehicles.
- [21] International Electrotechnical Commission. 2011. Plugs, socket-outlets, vehicle connectors and vehicle inlets - Conductive charging of electric vehicles: IEC 62196.
- [22] International Electrotechnical Commission. 2017. Electric vehicle conductive charging system: IEC 61851.
- [23] International Electrotechnical Commission. 2018. Road vehicles – Vehicle to grid communication interface: ISO 15118.
- [24] Alma L. Juarez-Dominguez (Ed.). 2008. *Feature Interaction Detection in the Automotive Domain: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*.
- [25] Alma L. Juarez-Dominguez, Nancy A. Day, and Jeffrey J. Joyce. 2008. Modelling feature interactions in the automotive domain. In *Proceedings of the 2008 international workshop on Models in software engineering*, Joanne Atlee (Ed.). ACM, New York, NY, 45.
- [26] Dirk O. Keck and Paul J. Kuehn. 1998. The feature and service interaction problem in telecommunications systems: a survey. *IEEE Transactions on Software Engineering* 24, 10 (1998), 779–796.
- [27] Willett Kempton and Jasna Tomić. 2005. Vehicle-to-grid power implementation: From stabilizing the grid to supporting large-scale renewable energy. *Journal of power sources* 144, 1 (2005), 280–294.
- [28] Jihyun Lee, Sungwon Kang, and Danhyung Lee. 2012. A Survey on Software Product Line Testing. In *Proceedings of the 16th International Software Product Line Conference - Volume 1 (SPLC '12)*. ACM, New York, NY, USA, 31–40.
- [29] Ivan do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana da Almeida. 2014. On strategies for testing software product lines: A systematic literature review. *Information and Software Technology* 56, 10 (2014), 1183–1199.
- [30] Urzti Markieg. 2017. Test Optimisation for Highly-Configurable Cyber-Physical Systems. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume B*. ACM, 139–144.
- [31] Bryan J. Muscedere, Robert Hackman, Davood Anbarnam, Joanne M. Atlee, Ian J. Davis, and Michael W. Godfrey. 2019. Detecting Feature-Interaction Symptoms in Automotive Software using Lightweight Analysis. In *SANER '19*, Xingyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, Piscataway, NJ, 175–185.
- [32] Sebastian Oster. 2005. Feature Model-based Software Product Line Testing. *International Workshop on Software Product Line Testing* 49, 12, 78–81.
- [33] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.
- [34] Brian Prasad. 1997. Analysis of pricing strategies for new product introduction. *Pricing Strategy and Practice* 5 (1997), 132–141.
- [35] pure systems. 2019. pure::variants: Variant Management with pure::variants. [www.pure-systems.com/products/pure-variants-9.html](http://www.pure-systems.com/products/pure-variants-9.html)
- [36] Jens Schmutzler, Christian Wietfeld, and Claus Amstrup Andersen. 2012. Distributed energy resource management for electric vehicles using IEC 61850 and ISO/IEC 15118. In *Vehicle Power and Propulsion Conference (VPPC)*. IEEE, 1457–1462.
- [37] Ken Schwaber and Mike Beedle. 2002. *Agile software development with Scrum*. Prentice Hall, Upper Saddle River, NJ.
- [38] Vanessa Stricker, Andreas Metzger, and Klaus Pohl. 2010. Avoiding Redundant Testing in Application Engineering. In *Software Product Lines: Going Beyond*, Jan Bosch and Jaejoon Lee (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 226–240.
- [39] T. F. Bowen, F. S. Dworack, C. H. Chow, N. Griffeth, G. E. Herman, and Y. - Lin (Eds.). 1989. *The feature interaction problem in telecommunications systems: Seventh International Conference on Software Engineering for Telecommunication Switching Systems*, 1989. SETSS 89.
- [40] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (2014), 1–45.
- [41] Len Woźniak and Paul Clements. 2015. How automotive engineering is taking product line engineering to the extreme. In *Proceedings of the 19th International Conference on Software Product Line*. ACM, 327–336.

# Feature-Oriented Contract Composition

Thomas Thüm  
TU Braunschweig, Germany

Alexander Knüppel  
TU Braunschweig, Germany

Stefan Krüger  
Paderborn University, Germany

Stefanie Bolle  
TU Braunschweig, Germany

Ina Schaefer  
TU Braunschweig, Germany

## ABSTRACT

A software product line comprises a set of products that share a common code base, but vary in specific characteristics called features. Ideally, features of a product line are developed in isolation and composed subsequently. Product lines are increasingly used for safety-critical software, for which quality assurance becomes indispensable. While the verification of product lines gained considerable interest in research over the last decade, the subject of how to specify product lines is only covered rudimentarily. One challenge is composition; similar to inheritance in object-oriented programming, features of a product line may refine other features along with their specifications.

In our work [1], we present a comprehensive discussion and empirical evaluation of *how to specify product lines* implemented by means of *feature-oriented programming*. In feature-oriented programs, implementation artifacts, such as methods, are distributed over the set of feature modules and subsequently composed together when the respective features are selected. Similar to this idea, contracts could be modularized, too, and are subsequently composed together with their respective methods. In particular, we investigate how refinement and composition of such specifications can be established and derive a notion of *feature-oriented contracts* comprising preconditions, postconditions, and framing conditions of a method (i.e., following the *design-by-contract* paradigm).

While both design by contract and feature-oriented programming have been hot research topics for more than two decades, their combination had rarely been explored. When features refine methods, an important question is whether refinement of their contracts is inevitable or not. However, unlike method composition where only the order of features is relevant, it seems that contract composition has to be handled differently according to certain scenarios. Consequently, a diverse set of composition techniques is required. In total, we identify and discuss six mechanisms to perform contract composition between original and refining contracts. Moreover, we identify and discuss desired properties for contract composition and evaluate which properties are established by which mechanism. As proof-of-concept and to enable larger evaluations, we developed tool support for feature-oriented contracts and their composition in FEATUREHOUSE and FEATUREIDE.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3342374>

We conducted an empirical evaluation, in which we specified 14 product lines by means of contracts. To evaluate product lines specified with feature-oriented contracts, we applied three strategies. First, we implemented five product lines and feature-oriented contracts from scratch. Second, we decomposed six existing, object-oriented programs, which were formally verified before, including their contracts into a product line. That is, we identified features of the program and separated them into feature modules. Third, we specified three existing product lines with feature-oriented contracts. Each of these creation strategies is a typical application scenario of employing feature-oriented contracts and may impose different requirements for contract-composition mechanisms.

We gained six insights from our work. First, the majority of contracts defined for product lines are not contained in all products (i.e., family-wide specification is not sufficient). Second, product-line specifications can be given by specifying each feature module and usually even without derivative modules (i.e., feature-based specification is sufficient). Third, most but not all method refinements establish behavioral subtyping, which means that the *Liskov principle* does not apply to feature specifications. Fourth, we identified that four of our six mechanisms were superior to all other mechanisms for certain contract refinements, and thus we conclude that these four mechanisms should be used in concert. Fifth, fine-granular contract refinements and alternative method introductions often cause specification clones. Finally, most contract refinements only refine the postcondition while the precondition and framing condition remain unchanged. In particular, only eleven out of sixty contract refinements modified the frame.

## CCS CONCEPTS

- Software and its engineering → Software product lines.

## KEYWORDS

Feature-oriented programming, Software product lines, Design by contract, Deductive verification, Formal methods

## ACM Reference Format:

Thomas Thüm, Alexander Knüppel, Stefan Krüger, Stefanie Bolle, and Ina Schaefer. 2019. Feature-Oriented Contract Composition. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342374>

## REFERENCES

- [1] Thomas Thüm, Alexander Knüppel, Stefan Krüger, Stefanie Bolle, and Ina Schaefer. 2019. Feature-oriented contract composition. *Journal of Systems and Software* 152 (2019), 83–107.

# Extraction of Configurable and Reusable Microservices from Legacy Systems: An Exploratory Study

Luiz Carvalho

Pontifical Catholic University of Rio de Janeiro  
Rio de Janeiro, Rio de Janeiro, Brazil  
lmcarvalho@inf.puc-rio.br

Alessandro Garcia

Pontifical Catholic University of Rio de Janeiro  
Rio de Janeiro, Rio de Janeiro, Brazil  
afgarcia@inf.puc-rio.br

Wesley K. G. Assunção

Federal University of Technology - Paraná  
Toledo, Paraná, Brazil  
wesleyk@utfpr.edu.br

Rodrigo Bonifácio

University of Brasília  
Brasília, DF, Brazil  
rbonifacio@unb.br

Leonardo P. Tizzei

IBM Research  
São Paulo, São Paulo, Brazil  
ltizzei@br.ibm.com

Thelma Elita Colanzi

State University of Maringá  
Maringá, Paraná, Brazil  
thelma@din.uem.br

## ABSTRACT

Microservices is an emerging industrial technique to promote better modularization and management of small and autonomous services. Microservice architecture is widely used to overcome the limitations of monolithic legacy systems, such as limited maintainability and reusability. Migration to a microservice architecture is increasingly becoming the focus of academic research. However, there is little knowledge on how microservices are extracted from legacy systems in practice. Among these limitations, there is a lack of understanding if variability is considered useful along the microservice extraction from a configurable system. In order to address this gap, we performed an exploratory study composed of two phases. Firstly, we conducted an online survey with 26 specialists that contributed to the migration of existing systems to a microservice architecture. Secondly, we performed individual interviews with seven survey participants. A subset of the participants (13 out of 26) dealt with systems with variability during the extraction, which stated that variability is a key criterion for structuring the microservices. Moreover, variability in the legacy system is usually implemented with simple mechanisms. Finally, initial evidence points out that microservices extraction can increase software customization.

## CCS CONCEPTS

• Software and its engineering → Software architectures; Maintaining software; Software product lines; Reusability; Software evolution.

## KEYWORDS

Microservice architecture, microservice customization, software variability, architecture migration

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336319>

## ACM Reference Format:

Luiz Carvalho, Alessandro Garcia, Wesley K. G. Assunção, Rodrigo Bonifácio, Leonardo P. Tizzei, and Thelma Elita Colanzi. 2019. Extraction of Configurable and Reusable Microservices from Legacy Systems: An Exploratory Study. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3336294.3336319>

## 1 INTRODUCTION

A legacy system, commonly found in industry, represents a long term massive investment. Despite of their business importance, legacy systems are difficult to extend and include innovation [5]. In addition, these systems usually have a monolithic architecture, with components tangled in a single unit, strongly connected and interdependent [20, 27]. Currently, many companies have been adopting microservice architectures to modernize monolithic legacy systems [6, 14, 17, 20, 28, 30]. Microservices are small and autonomous services that work together [23]. The benefits of adopting microservices are: reduced effort for maintenance and evolution, increased availability of services, ease of innovation, continuous delivery, ease of DevOps incorporation, facilitated scalability of parts with more demand, etc [20, 27].

Service-based architectures must meet some attributes to satisfactorily fulfill their purpose [10]: (i) scalability, enabling optimization in the use of hardware resources to meet high demand services; (ii) multi-tenant efficiency, offering transparency in the use of shared services, since the service should optimize the sharing of resources and also isolate the behavior of different tenant; and (iii) variability, where a single code base provides common and variable functionalities to all tenants. Microservices are known to work well with the first two attributes [28]. However, the literature is scarce in relation to the use of variability. A possible explanation for the lack of discussion may be the fact that this technology was initially conceived in the industry and only in the last years have had attention of academia<sup>1</sup>.

Configurable systems are widely developed in the industry to meet demands related to different types of hardware, platforms, serving diverse customers or market segments, etc [29]. In this context, we need to understand how functionalities should be modularized and customized as microservices to fulfill customer needs.

<sup>1</sup><https://martinfowler.com/articles/microservices.html>

The goal of this paper is to investigate the importance of variability on the process of extracting microservices from monolithic legacy systems in industry. To this end, we analyzed 26 survey responses and seven interviews with practitioners. Among the 26 participants, 13 dealt with variability during the extraction and they stated that variability was a key criterion for structuring the microservices. Moreover, we observed an increase of requests for customization after microservices extraction that initially were not configurable. In this way, initial evidence points out that the microservices extraction can increase software customization.

This paper is organized as follows. Section 2 presents the background. Section 3 details our study. Results and discussion are in Section 4. Sections 5 and 6 address threats to validity and related work, respectively. Section 7 concludes the paper.

## 2 BACKGROUND

This section presents the main concepts involved in this paper, such as microservices, customization, and variability.

### 2.1 Microservices

A microservice is expected to have fine granularity [11] and be autonomous: (i) it should consist of a service highly independent from others, and (ii) it should enable an independent choice of technologies. For communication, microservices architecture commonly adopted lightweight protocols.

Not rarely, microservices are not developed from scratch, but they result from the migration of existing systems [14, 20, 23]. However, the migration from an existing system to microservices is perceived as challenging by developers who have experienced it [20, 27]. Previous studies indicate that developers use the source code of the existing system in migration to microservice architecture [15]. In the same way, approaches were proposed using criteria observed in source code [18, 21], generally using coupling and cohesion criteria. Others approaches recommend observing the database schema [13, 23]. However, there is a lack of the comprehension of variability usefulness in the migration process when the existing system is a configurable system or software product line.

### 2.2 Customization and Variability

Service-oriented architectures must allow tenant-specific configuration and customization. The tenant-specific adaptations may affect all layers of the application, from functional requirements to database schema. Furthermore, tenants do not only have different requirements regarding functional properties, they can also require different non-functional requirements of service properties [22].

To achieve such customization, industry must deal with software variability, which is the ability of a system or an asset to be adapted for using in a particular context [3]. Variability can be viewed as consisting of two dimensions [8]. The space dimension is concerned with the use of software in multiple contexts. The time dimension is concerned with the ability of software to support evolution and changing requirements in its various contexts.

## 3 EXPLORATORY STUDY DESIGN

The goal of our exploratory study is to better understand the migration process to microservice architecture and how software

variability is useful in the decision-making process. In what follows, we show in Subsection 3.1 the research questions and their motivations. Subsection 3.2 presents the two phases (survey and interview), population, and sampling. In addition, Subsection 3.3 shows the instruments used in the survey and interview.

### 3.1 Research Questions

Based on the aforementioned goal, we intend to answer the following research questions:

**RQ1** How important is variability in the migration process to the microservice architecture?

- **RQ1.1** Did the original systems contain some sort of customization prior to the migration?
- **RQ1.2** What mechanisms were used to implement variability prior to the migration to microservice architecture?
- **RQ1.3** Do developers consider variability as useful criteria in the migration process to microservice architecture?

**RQ2** How is variability present after migration to the microservice architecture?

- **RQ2.1** Does microservices extraction increase software customization?
- **RQ2.2** What mechanisms are used to software customization in microservice architecture?

In summary, for RQ1 we want to investigate the relevance of variability and its mechanisms to support and guide the migration of configurable systems to a microservice architecture. RQ2 aims to understand how variability is present after the migration to a microservice architecture, since there is a lack of understanding about how configurable systems coexist with the microservice architecture.

### 3.2 Study Phases, Population and Sample

Our exploratory study is composed of two phases: (i) a survey with specialists, and (ii) interviews with specialists who answered the survey. The target of our survey are practitioners and industrial researchers with a background in migrating existing systems to microservices. We used the results of three mapping studies [1, 16, 24] to identify an initial target population. In addition, we performed a snowballing search [31] in studies that cited the referred mapping studies. For performing this search, we used Google Scholar<sup>2</sup>.

Our strategy to contact specialists was to invite the authors of the papers found in the three mapping studies to participate in the survey. We requested that they submit the survey to the subjects of the empirical studies. After execution, our search plan resulted in the recruitment of 90 subjects. The survey was executed from January to March 2019. From the subjects recruited, we collected answers from 26 participants, resulting in a participation rate of 29%. Of which 13 participants also answered our questions related to software variability. To carry out deep analysis, we invited the survey participants to an interview. We conducted 7 interviews.

<sup>2</sup><https://scholar.google.com/>

### 3.3 Instrumentation

**Survey.** We organized the survey into two groups. The first group is composed of questions for characterizing the participants. We asked the academic background, development experience, and position in the current job. We also inquired about their background in migrating existing systems to microservices.

The second group of questions has the objective of perceiving the usefulness of variability, in the extraction of microservices. For this group of questions, we used a five-point Likert scale associated with the levels of usefulness for variability, from the least (1) to the most (5). We also asked the participants to justify their answers. To avoid misinterpretations, we provided in the questionnaire a definition of variability, which is the ability to derive different products from a common set of artifacts [2].

**Interview.** The goal of this phase was to understand the post characteristics of the migration process. The interviews were conducted using video conference tools. Each interview was conducted by an interviewer and a scribe that took notes. For the execution of the interview, we chose a semi-structured approach. That is, questions that answers can be quantified (structured) and suggest the theme (unstructured) [26].

Regarding the questions, the participants were first questioned with open and general questions about more high-level themes, in an unstructured way. For example, explaining about the existing system that was/is being migrated to microservice architecture. After that, we inquired them with quantitative questions. For example, the number of microservices extracted. This approach was chosen by observing that survey participants had an experience mean of 15.77 years in software development. Moreover, participants have been involved in the migration process to the microservice architecture. In the questions, we investigated the themes: (i) existing systems, (ii) migration process, (iii) variability, and (iv) tools.

## 4 RESULTS AND ANALYSIS

In this section, we present the results and their analysis. The first phase of our study (survey) offers a better understanding about the usefulness of variability.

In a previous study, we observed how relevant are seven criteria during migration to microservices architecture. However, the previous study did not perform an analysis of variability [9]. This happened because of the low number of participants with expertise on migrating some existing system with variability to a microservice architecture. In this present work, the greater number of survey answers allowed us made analyses about variability in the first phase. Moreover, the second phase of study (interview) provided initial results about the post-migration process to microservice architecture. Among them, the request for increased customization of the microservices in order to deal with different customers or groups, in which, before the migration process was not made.

### 4.1 Participants Characterization

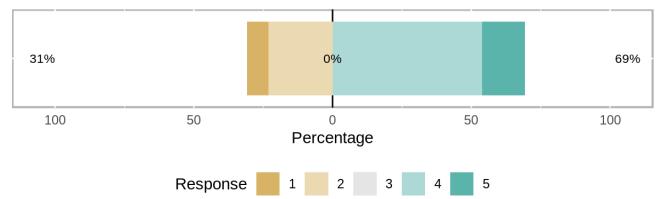
Survey respondents experience may be evidenced by previous and ongoing activities with the migration process to microservice architecture. The vast majority of the respondents had already concluded their participation in at least two migration process (mean of 2.5

and median of 2.0) and most of them are actively participating in at least one project (mean of 1.3 and median of 1.0) where the microservices extraction is currently underway. Besides that, participants observed at least one process (a mean of 1.5 and median of 1.0) and they are observing at least one migration (a mean of 1.5 and median of 1.0). The respondents have extensive experience in software development. The time in years the participants have been developing software is considerably high: (i) the mean is 15.77 years and median is 15 years, and (ii) with a maximum of 35 years and a minimum of 3 years. Our sample of respondents is diversified in terms of the roles that they play in their employment: developers (42%), architects or engineers (23%), team leaders (19%), and industry researchers (19%).

### 4.2 Variability

From the 26 participants, 50% have never considered variability. For these respondents, the domain they work does not require the ability to derive different products from a common set of artifacts [2]. Their intention was to specifically migrate a single software system to a microservice architecture. However, the other half of the participants have answered that their existing systems, which were the target of microservices extraction, had variability. Thus, these respondents were able to answer about the usefulness of variability during the migration process to a microservice architecture.

Among the participants with previous experience to answer on the usefulness of this criterion (13 out of 26) (see Figure 1): (i) 31% considered it as not relevant by assigning values of 1 or 2 in the Likert scale, and (ii) 69% of the participants that migrated systems with variability considered it as useful or very useful. There is no response with moderate usefulness. Among those that considered variability important, one participant said: “*It was useful to identify the variabilities and features of each product*”. Other participant said that “*There are 3 systems that will be affected by the migration and we need to ensure the differences between them and how these differences can be handled by the migration process*”. In this way, most of the participants that migrated systems with variability consider variability useful or very useful in the migration process to the microservices architecture. However, previous approaches to microservices extraction do not make use of variability criterion.



**Figure 1: The usefulness of variability for the migration process to microservices architecture**

Regarding the variability implementation approaches, the most common ones are those shown in Table 1. In the last column of the table, we present the proportion in which they were cited.

Version control was the most common variability implementation approach used in existing systems, that is, in systems before the migration to microservice architecture. One might consider this is the most simplistic approach as it leads to the management

**Table 1: Variability Implementation Approaches**

Approach	Percentage
Version control	77%
Design patterns	62%
Framework	62%
Components	54%
Parameters	46%
Build systems	38%
Aspect-oriented programming	31%
Feature-oriented programming	15%
Pre-processor	8%

of copy and paste of different products. For example, without the need of extending the programming language being adopted as in feature-oriented programming. Approaches more sophisticated are less common ones, like the use of aspect-oriented programming and feature-oriented programming. The mean of variability implementation approaches used is 4.5. All the respondents mentioned more than one approach to implement variability.

The survey results enable us to answer **RQ1** as follows:

**RQ1.1:** 50% of the participants answered that at least one of their existing systems (from which microservices were extracted) had variability.

**RQ1.2:** Version control is the most used mechanism (77%) to implement variability in those existing systems.

**RQ1.3:** 69% of the participants that migrated systems with variability considered that variability is a useful or very useful criterion during the migration process.

### 4.3 Microservice Customization

During the interview phase, we inquired the participants whether the extraction allowed that some microservices could be used in different contexts. To our surprise, four (57%) of the interviewed participants said that some customization was required after this migration process to microservice architecture. The customization was needed to attend requests of different groups inside the same enterprise or customers of the software system. The four cases previously answered in the survey that the systems prior to migration to microservices had no variability. This seems to be an indication that the process of migration to microservices leverages the customization of the single system by increasing modularity and more interfaces available to its customers. Common ways to manage these new requirements for dealing with customization of the software systems are reported in the following paragraphs.

Regarding the other three participants (43%) of the interview, two of them reported in both the survey and the interview that: (i) the system has variability before the migration to microservice architecture, (ii) and the system continued to have variability after the migration. In one of these cases, the variability in the migrated system relied on design patterns and parameters in the interface to identify a tenant [22]. In the other case, the migration process is still underway. However, it is expected that some microservices will not exist or have their behavior drastically modified in different usage contexts. In the third case, which there was no variability

before or after the migration, the software system was completely developed from scratch; then, the system was migrated afterward to satisfy a single customer. It was not used by different groups of the enterprise.

**RQ2.1:** Initial evidence points out that the migration process to a microservices architecture increases the customizations of the system.

During the interviews, four participants reported that the migration to microservices was not made with the goal of turning the system more easily customizable. However, after the migration, customization requests emerged by different clients or groups within the same enterprise.

Among the four cases that reported this growth for post-migration customization, three of them described which approaches were adopted to implement variability in a microservice architecture. One of the participants reported that the need for customization emerged after the microservices extraction; however, he did not observe or participate in the process of implementing or managing the customizations.

We describe below the three implementation approaches employed to deal with the customization required after the microservices extraction: *Copy and Paste*, *Big Interface*, and *Filtering to the Different Platforms*.

*Copy and Paste* was adopted after the migration process to microservice architecture by one of the participants. That happened when customizations in the interface of extracted microservices began to be requested by different groups within the same enterprise. In this case, it was decided to copy the microservices implementation and perform the customization. Thus, the copied microservice coexists with the original one (inclusive at runtime). Changes are performed in both microservices by the original developers when maintenance is required in mandatory features. It was also reported that code of test cases is submitted to the same copy and paste process and it is managed in the same way. This approach is often used in the industry as it promptly promotes reuse of verified functionalities, without requiring high upfront investment and achieving short-term benefits, as reported by Dubinsky *et al.* [12].

*Big Interface* was also an adopted solution. In this case, certain consumers of the microservice APIs needed additional information, or the same information in different formats. To fulfill this demand, the developers just included all required information in already existing APIs. However, it should be noted that this solution raises well-known issues related to the big interface problem [7].

*Filtering to the Different Platforms* was an approach that uses an intermediary microservice between other microservices and consumers, similar to a gateway microservice. This intermediary microservice has the sole purpose of filtering the outputs for specific platforms. The case described by the participant had three different platforms, namely mobile, web, and desktop, that need customized information from the same microservice. For example, when a consumer uses a mobile platform, the user interface returns more targeted and lean responses to these devices. That is, it was a filter of the original outputs from target microservice interface with its customization for the different platforms. This pattern seems to

be a form of resolution to mitigate problems from the previously presented approach.

In addition to the approaches to deal with the customization that emerged after the migration process to the microservice architecture, the interviewed participants were also inquired about integration between extracted microservices and the legacy system. We observed that, in five cases, where the migration process was completed by the interview date, four of them extracted the microservices in an incremental process. That is, some microservices were extracted and integrated with the legacy system and the resulting behavior was observed. In this integration, it was common to use feature toggles or similar mechanisms to switch between the legacy system and the integration between extracted microservices. A feature toggle basically is a variable used in a conditional statement to guard code blocks, with the aim of either enabling or disabling the feature code in those blocks for testing or release [4, 25]. This was used in a production environment for the problems reported or observed by the team monitoring this transition. This approach allowed a fast return to a stable version (legacy system). In general, some microservices with database access (usually a key-value database) was used as a filter to choose the switch between using only the legacy system and the extracted microservices integrated with the legacy system. One of the interviews reported the use of this switching strategy between both versions for a small portion of their users. 43% of the survey respondents reported the employment of specific steps related to the integration between the legacy system and the extracted microservices. Besides that, the effort required in most cases was medium or high.

**RQ2.2:** *Three approaches employed to implement the customization required after the microservices extraction are (i) Copy and Paste, (ii) Big Interface and (iii) Filtering to the Different Platforms. Microservices are usually extracted in an incremental process. Moreover, feature toggles (or similar mechanisms) are used to switch between the legacy system and the extracted microservices integrated with the legacy system.*

## 5 THREATS TO VALIDITY

The first threat is related to the number of questions in the survey, which might discourage the subjects' participation. To alleviate this threat we conducted an expert review with two specialists [19]. After considering their feedback, we conducted a pilot study with four subjects. In this pilot, we observed an acceptable participation rate.

Another threat concerns the number of valid respondents in the survey. This sampling process resulted in the recruitment of 90 individuals. A participation rate of 29% is considered good for online surveys of this kind, which usually ranges from 3% to 10%. In addition, most of the survey participants declared to have significant experience in migrating systems to microservice architecture (see Section 4.1). The quality of subjects was reached due to the fact we followed a formal recruitment strategy (see Section 3.2).

As far as the interview is concerned, a threat affects the process of collecting data during the interviews. To deal with this threat we asked the participants permission to record the interviews for

future analysis and transcriptions. In addition, all interviews were performed by an interviewer, which presented the questions; and a scribe responsible for taking notes, analyzing the respondent behavior, and asking additional questions.

## 6 RELATED WORK

Francesco *et al.* [15] interviewed and applied a questionnaire to developers. Their goal was to understand the performed activities, and the challenges faced during the migration. They reported what are the existing system artifacts (e.g., source code and documents) the respondents used to support the migration. The main reported challenges were: (i) the high level of coupling, (ii) the difficulty of identifying the service boundaries, and (iii) the microservices decomposition. However, they did not specifically analyze the usefulness of variability criterion addressed in our survey.

Taibi *et al.* [27] also conducted a survey with the objective of elucidating motivations that led to the microservices migration process and what were the expected returns. The main motivations were the improvement of maintainability, scalability, and delegation of team responsibilities. In addition, difficulties were cited in this process, such as decoupling from the monolithic system, followed by migration, and splitting of data in legacy databases.

## 7 CONCLUSIONS

This paper presented an exploratory study composed of two phases. In the first phase, a survey was applied to specialists experienced in the migration process to microservice architectures. In this survey, we also inquired about the usefulness of variability and implementation mechanisms. In the second phase, we asked an interview with survey participants. We asked about customization requests after the microservices extraction, and the mechanisms used by the participants to deal with post-migration customization.

The results revealed that half of the participants dealt with systems with variability during the migration. Among these practitioners, two-third considered variability as useful or very useful to extract of microservices. We also observed initial evidence that microservices extraction can increase software customization, mainly because some users requested for customization after the microservices extraction. The most common approaches to implement customization in extracted microservices are copy and paste, big interface, and filtering to different platforms. Moreover, feature toggles (or similar mechanisms) are commonly used to switch/integrate the legacy system and the extracted microservices.

Our study is still ongoing. So, for future work we expect to have more responses to our survey and more interviewees. In this way, we will be able to draw additional analyses on other useful criteria for extracting configurable microservices from monolithic legacy systems.

## ACKNOWLEDGMENTS

This research was funded by CNPq grants 434969/2018-4, 312149/2016-6, 408356/2018-9 and 428994/2018-0, CAPES grant 175956, FAPERJ grant 22520-7/2016.

## REFERENCES

- [1] N. Alshuqayran, N. Ali, and R. Evans. 2016. A Systematic Mapping Study in Microservice Architecture. In *International Conference on Service-Oriented Computing and Applications (SOCA)*. 44–51.
- [2] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated.
- [3] Felix Bachmann and Paul C Clements. 2005. *Variability in software product lines*. Technical Report CMU/SEI-2005-TR-012. Carnegie Mellon University - Software Engineering Institute.
- [4] Len Bass, Ingo Weber, and Liming Zhu. 2015. *DevOps: A software architect's perspective*. Addison-Wesley Professional.
- [5] Jesús Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. 1999. Legacy Information Systems: Issues and Directions. *IEEE Software* 16, 5 (Sept. 1999), 103–111.
- [6] A. Bucciarone, N. Dragoni, S. Dusdhar, S. T. Larsen, and M. Mazzara. 2018. From Monolithic to Microservices: An Experience Report from the Banking Domain. *IEEE Software* 35, 3 (2018), 50–55.
- [7] Robert C. Martin. 2002. *Agile Software Development, Principles, Patterns, and Practices* (1st ed.). Pearson.
- [8] Rafael Capilla, Jan Bosch, and Kyo-Chul Kang. 2013. *Systems and Software Variability Management: Concepts, Tools and Experiences*. Springer Publishing Company, Incorporated.
- [9] Luiz Carvalho, Alessandro Garcia, Wesley K. G. Assunção, Rafael de Mello, and Maria Julia de Lima. 2019. Analysis of the Criteria Adopted in Industry to Extract Microservices. In *International Workshop on Conducting Empirical Studies in Industry and International Workshop on Software Engineering Research and Industrial Practice (CESSER-IP)*. IEEE Press, Piscataway, NJ, USA, 22–29.
- [10] Frederick Chong and Gianpaolo Carraro. 2006. Architecture strategies for catching the long tail. *MSDN Library, Microsoft Corporation* (2006), 9–10.
- [11] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. *Microservices: Yesterday, Today, and Tomorrow*. Springer International Publishing, Cham, 195–216.
- [12] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An exploratory study of cloning in industrial software product lines. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34.
- [13] D. Escobar, D. Cárdenas, R. Amarillo, E. Castro, K. Garcés, C. Parra, and R. Casallas. 2016. Towards the understanding and evolution of monolithic applications as microservices. In *XLI Latin American Computing Conference (CLEI)*. 1–11.
- [14] Susan Fowler. 2016. *Production-Ready Microservices* (1st ed.). O'Reilly Media.
- [15] P. Di Francesco, P. Lago, and I. Malavolta. 2018. Migrating Towards Microservice Architectures: An Industrial Survey. In *International Conference on Software Architecture (ICSA)*. 29–299.
- [16] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. 2019. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software* 150 (2019), 77 – 97.
- [17] J. Gouigoux and D. Tamzalit. 2017. From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture. In *International Conference on Software Architecture Workshops (ICSAW)*. 62–65.
- [18] W. Jin, T. Liu, Q. Zheng, D. Cui, and Y. Cai. 2018. Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering. In *International Conference on Web Services (ICWS)*. 211–218.
- [19] Johan Linaker, Sardar Muhammad Sulaman, Rafael Maiani de Mello, Martin Höst, and Per Runeson. 2015. Guidelines for Conducting Surveys in Software Engineering. (2015).
- [20] Welder Luz, Everton Agilar, Marcos César de Oliveira, Carlos Eduardo R. de Melo, Gustavo Pinto, and Rodrigo Bonifácio. 2018. An Experience Report on the Adoption of Microservices in Three Brazilian Government Institutions. In *XXXII Brazilian Symposium on Software Engineering (SBES)*. ACM, New York, NY, USA, 32–41.
- [21] G. Mazlami, J. Cito, and P. Leitner. 2017. Extraction of Microservices from Monolithic Software Architectures. In *International Conference on Web Services (ICWS)*. 524–531.
- [22] Ralph Mietzner, Andreas Metzger, Frank Leymann, and Klaus Pohl. 2009. Variability Modeling to Support Customization and Deployment of Multi-tenant-aware Software As a Service Applications. In *Workshop on Principles of Engineering Service Oriented Systems (PESOS)*. IEEE Computer Society, Washington, DC, USA, 18–25.
- [23] Sam Newman. 2015. *Building Microservices* (1st ed.). O'Reilly Media.
- [24] Claus Pahl and Pooyan Jamshidi. 2016. Microservices: A Systematic Mapping Study. In *International Conference on Cloud Computing and Services Science (CLOSER)*. 137–146.
- [25] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C. Rigby, and Bram Adams. 2016. Feature Toggles: Practitioner Practices and a Case Study. In *13th International Conference on Mining Software Repositories (MSR)*. ACM, New York, NY, USA, 201–211.
- [26] C. B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering* 25, 4 (July 1999), 557–572.
- [27] D. Taibi, V. Lenarduzzi, and C. Pahl. 2017. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing* 4, 5 (2017), 22–32.
- [28] Leonardo P. Tizzei, Marcelo Nery, Vinícius C. V. B. Segura, and Renato F. G. Cerqueira. 2017. Using Microservices and Software Product Line Engineering to Support Reuse of Evolving Multi-tenant SaaS. In *21st International Systems and Software Product Line Conference (SPLC)*. ACM, New York, NY, USA, 205–214.
- [29] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-condition Simplification in Highly Configurable Systems. In *37th International Conference on Software Engineering (ICSE)*. IEEE Press, Piscataway, NJ, USA, 178–188.
- [30] Coburn Watson, Scott Emmons, and Brendan Gregg. 2015. *A Microscope on Microservices*. <http://techblog.netflix.com/2015/02/a-microscope-on-microservices.html>
- [31] Claes Wohlin. 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, New York, NY, USA, Article 38, 10 pages.

# Journal First Presentation of a Comparative Study of Workflow Customization Strategies: Quality Implications for Multi-Tenant SaaS

Majid Makki, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen  
firstname.lastname@cs.kuleuven.be  
imec-DistriNet  
Department of Computer Science, KU Leuven  
3001 Heverlee, Belgium

## ABSTRACT

Multi-tenant Software-as-a-Service (SaaS) applications share a single runtime instance among multiple customer organizations (tenants). To account for differences in tenant requirements, they have to support run-time customization. The latter turns these types of applications into a dynamic software product lines involving a wide range of software artifacts such as user interfaces, databases, web-services and business process or workflow definitions.

This paper analyzes and compares the quality implications of different business process customization strategies for multi-tenant SaaS applications. The customization strategies are selected from an existing survey and the comparison criteria are derived from two essential characteristics of SaaS: it is (i) a business model aiming at “economies of scale”, and (ii) a software delivery model with specific automation requirements.

The comparative study shows that there is no single best strategy, and provides SaaS architects with support for making appropriate trade-off decisions when adopting a workflow customization strategy. As a by-product of this study, a number of points for future improvement and innovations in existing workflow technology are identified, which are exclusively relevant for multi-tenant SaaS applications.

## CCS CONCEPTS

- **Software and its engineering** → *Software organization and properties; Software creation and management; Software product lines.*

## KEYWORDS

Multi-tenancy, Software-as-a-Service, Workflow automation, Functional customization, Software quality

### ACM Reference Format:

Majid Makki, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. 2019. Journal First Presentation of a Comparative Study of Workflow Customization Strategies: Quality Implications for Multi-Tenant SaaS. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19)*, September 9–13, 2019, Paris, France. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342371>

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7138-4/19/09.

<https://doi.org/10.1145/3336294.3342371>

'19), September 9–13, 2019, Paris, France. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342371>

## Description

This is an extended abstract of the paper titled “A Comparative Study of Workflow Customization Strategies: Quality Implications for Multi-tenant SaaS” published in *Journal of Systems and Software* [1].

## ACKNOWLEDGMENTS

This research is partially funded by the Research Fund KU Leuven, the ADDIS research program funded by KU Leuven GOA, and the DeCoMAdS and DISSECT SBO strategic research projects.

## REFERENCES

- [1] Majid Makki, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. 2018. A comparative study of workflow customization strategies: Quality implications for multi-tenant SaaS. *Journal of Systems and Software* 144 (2018), 423–438.

# App Variants and Their Impact on Mobile Architecture: An Experience Report

Marc Dahlem

Insiders Technologies GmbH

Kaiserslautern, Germany

M.Dahlem@insiders-technologies.de

Ricarda Rahm

Insiders Technologies GmbH

Kaiserslautern, Germany

R.Rahm@insiders-technologies.de

Martin Becker

Fraunhofer IESE

Kaiserslautern, Germany

Martin.Becker@iese.fraunhofer.de

## ABSTRACT

In order to raise the awareness of industrial practitioners and researchers regarding specific PLE-related issues and approaches, this paper shares some experiences made by Insiders Technologies regarding the development and provisioning of mobile app variants and the impact of variability on the app architecture. Using the smart MOBILE app product line as an example, the paper characterizes the mobile app market, identifies key variant drivers, introduces influential technologies and their constraints, and discusses viable tactics to support adequate variability in the architecture of a mobile app.

## CCS CONCEPTS

- Software and its engineering → Software product lines.

## KEYWORDS

mobile application, mobile app development, software architecture, software product line

### ACM Reference Format:

Marc Dahlem, Ricarda Rahm, and Martin Becker. 2019. App Variants and Their Impact on Mobile Architecture: An Experience Report. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3336294.3336320>

## 1 INTRODUCTION

Since the early days of software product line engineering (PLE), the telecommunication domain has been a promising area for strategic reuse approaches. Various convincing success cases on how product line engineering helped mobile phone companies provide more product variants in less time with higher quality have been published at SPLC and beyond [3, 7].

While there is a strong consensus on the benefits of PLE, it remains challenging for organizations to identify methods and techniques that are applicable for their particular context, to adapt these methods and techniques in order to address the specific needs of their domain, and to integrate them with their current practices, tools, and standards [11].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336320>

The rise of smartphones and the corresponding software ecosystems has revolutionized the mobile phone market and the way software applications are provided to phone users. Unfortunately, only few experiences have been shared since then on specific variability-related challenges and viable approaches in the mobile app domain.

The *contributions* of this paper are a characterization of the mobile app domain, a discussion of the pivotal variant drivers, an overview of influential technologies and their constraints, as well as a discussion of viable tactics to support adequate variability in the architecture of a mobile app.

This paper aims to share experiences and possible solutions with those who have not yet gained much practical knowledge in the field of mobile applications.

## 2 RELATED WORK

In mobile app development, one of the biggest variant driver is that an app must be developed for multiple platforms [9]. This has not changed significantly in recent years. Therefore, different solutions have been proposed to deal with this situation. One idea is to no longer develop the app natively, but to use a hybrid variant in which rather a web application is built instead, which is then embedded into an app [5]. However, this kind of development has some drawbacks: For example, from the website, it is hardly possible to access the hardware of the device which could reduce the user experience. [1]. A more promising approach is to apply Product Line Engineering to product lines of mobile apps as well [4]. To this end, the handling of variability is discussed by Jørgensen et al. [8], who dealt with variable apps from the banking industry. In particular, the variability of branding, reflected in colors, images, and texts, was examined. Rosa et al. [13] investigated reuse in the context of mobile systems. They have built a software infrastructure that dynamically and automatically selects software components so that an app can be built. Flexibility, reusability and loose coupling are listed as the most important properties. The architecture of mobile apps was studied by Bagheri et al [2]. In their work, they describe the design and construction of mobile apps. To do this, they consider the Android platform exclusively and describe characteristics they found in the ecosystem of Android apps. They also compare how the design of architecture is proposed in the literature and how it has been implemented in the Android apps they studied. Fußberger et al. [6] have researched variability in the Android operating system. Their paper focuses on the usage of different variability mechanisms (e.g., Conditional Compilation) in the Android source code and build system.

### 3 CHALLENGES OF MOBILE APPLICATION DEVELOPMENT

In order to set the scene, this section introduces some specific challenges faced by mobile app developers during software development. The study of Joorabchi et al. [9] shows that the majority of mobile application developers see a problem in developing functionally similar software for *several operating systems* as well as for *different hardware platforms*. Currently, several operating systems exist for mobile devices, such as iOS or Android, and even within these subsets there are several hardware platforms. Also, users often use *different versions* within the different operating systems, for which the applications require appropriate adaptations. Thus, the operating system becomes one of the largest variant drivers, which also has a huge effect on other variabilities. Additionally, the fast changing updates of hardware and operating systems requires the development process to be continuously adapted and kept up-to-date.

Another challenge is *reuse of source code*, especially across platforms. The reason given for this is the lack of portability of code across different platforms, which is due to *different programming languages* such as Objective C or Swift on iOS or Java and Kotlin on Android. Some of the respondents of the study [9] stated that it would therefore be more useful to rewrite the applications for the different operating systems from scratch, since attempting to port software parts either harms quality or is not possible at all. This illustrates the profound effects of the variant driver of the operating system.

The operating systems also differ in terms of *user interfaces and user experience*. In addition, the expectations of the users of the different platforms are different, as well as existing APIs or SDKs and tool support. This problem also includes *non-standardized human computer interaction guidelines* for different platforms. An application should provide the same functions across platforms and appear similar to the user, but there is no generic design for all platforms. Thus the conflict arises that the applications should match to the look and feel of the respective platform, but should behave in same way across the different platforms. Together this results in several small variation points within the large variant driver of the operating system, whose variations contain, for example, the different SDKs or the design guidelines of the respective user interfaces. In some companies, development teams are separated by operating system [4] so that they can provide expertise for the respective requirements of the different systems. The applications are thus developed in isolation from each other, adapted to the respective systems.

The different language concepts and platforms pose a problem especially for the development of native applications that are executed directly on the operating system of the terminal [4, 9]. Native applications are developed directly for the device and use all the technical possibilities of the respective platform. In order to avoid the challenges of multiple platforms, some companies are developing hybrid applications. These are not executed directly on the device, but on a web server and are embedded into a native application displayed to the user, similar to a web browser. However, this type of development has some drawbacks. From the website, for example, it is *hardly possible to access the hardware of the device*,



**Figure 1: Product standard and derived product**

which affects the user experience, since the application feels more like a website [1].

All this leads to a *multiplication of costs, effort and budget* according to the number of variant drivers of the supported platforms. This multiplication affects not only cost, but also time-to-market, through the phases from design to maintenance.

In general, not all paradigms established in conventional software development are valid in mobile development due to the specific challenges [4]. In software development, the principle of reuse is often used to efficiently develop new software, which now should be considered in the mobile application domain.

## 4 SMART MOBILE APP PRODUCT LINE

### 4.1 Context

The business areas of Insiders Technologies GmbH (Insiders for short) are development and marketing of software products, consulting and introduction of methods and technologies for document management, as well as design and implementation of adaptive knowledge-based systems and corporate memories.

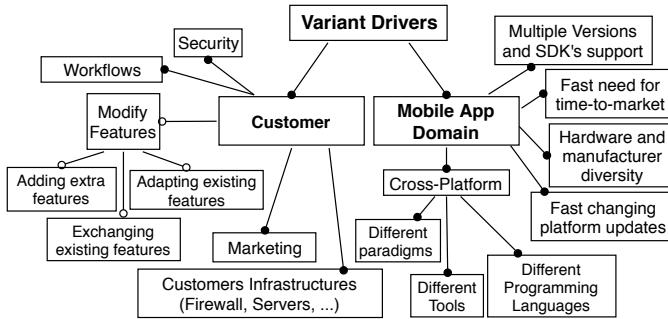
This paper places particular emphasis on looking at Insiders' portfolio of mobile applications. The company markets a white-label app to customers from the insurance, banking, and human resources domains. Among others, the application offers the possibility of document recognition and the ability to submit documents to the company.

According to a modular principle, a customer can choose different features, which are embedded in tiles presented on the main screen. In addition to predefined functions or custom behavior, a customer can also choose a website to be provided, as suggested in the hybrid approach.

To demonstrate how this looks like, figure 1 shows the product standard on the left and an example of a derived end product on the right. Four different main functions have been added to the main screen, the colors and icons have been adapted, and the naming has changed. This adaptions are part of the branding of the app and are done individual for each customer. This customizations are an important variant driver in the software and therefore, this must be considered in designing a infrastructure of the product standard.

## 4.2 Overview of Typical Variant Drivers and Impact on System Structure

As shown in Figure 2, the variant drivers of the product can be subdivided into two groups. On the one hand variant drivers are created by the domain; on the other hand, many variants are driven by the various customers.



**Figure 2: Variant drivers of the domain and those of the customers**

The variants which are driven by the mobile app domain, such as multiple platforms or the consequences of fast changing updates are mostly summarized in chapter 3. In addition, it was also important to analyze how customers' projects differ and especially, where in the product individual adjustments had to be applied the most.

From a product-specific point of view, adding, modifying, and exchanging features is a major variant driver. Also, adapting the product to the customer's infrastructure is also part of application engineering and can be very time-consuming, depending on the requirements. Although the customers are from similar domains, often entire workflows must be changed, which affects the implementation massively. To show an example here: if a document is sent to the customer, there are different processes that can happen afterwards. Some customers want to give the user a status update, others want to respond directly to the submission, others only accept the document. Another aspect that leads to different variants is system security, since each customer has different demands regarding the security of their system. As some customers belong to the eHealth sector, some special data protection regulations must be observed that are less relevant, for example, for customers from other domains. This requires, in particular, certain security measures, such as specific encryption methods. In general, these can change by customer, so it should be considered in the implementation that these need to be easily interchangeable. Montenegro et al. [10] have identified which aspects have to be considered when implementing the security for Android apps.

Some of these issues, such as security, not only affect the app but also the backend. Others, such as marketing, are more likely to affect the UI and some are not really predictable, such as adapting to the customer's infrastructure. All these issues must be considered when designing a new framework.

## 4.3 Current Framework Architecture

Originally, the product line was created from a single product which contained document capturing and a user management. Based on this, we received new customer inquiries for functionally similar products. First, our approach was to simply clone the product for the new customers and to add or customize required features.

As a matter of course, this resulted in many different product variants, as there was no architecture with customization support. Adaptations and additions of features were getting more difficult to implement and all the drawbacks of cloning became apparent. At some point it was decided to develop a new architecture, internally called "smart MOBILE Framework". It was especially designed to fulfill the support of adding and exchange individual features, the possibility to adapt existing features for customers without influencing other solutions and as feasible as possible to support cross-platform development.

The basic idea of the architecture is a modular concept, that supports all the aforementioned features. It was inspired by the idea of 'micro-services' [12], where every service has its own, very small responsibility and where all services communicate via predefined interfaces and APIs. Such architectures are commonly used (e.g. by Amazon, Netflix [14]) in order to split large software into small independent pieces to reduce complexity and make the implementation details (programming language, tools etc) independent of each service, but still allow a very good code reuse. Furthermore, as all services only trust the promises of the API's and interfaces of other services, such small pieces can easily be replaced by a new implementation that is still compliant with the API.

Normally, micro-services are hosted as stand-alone small server instances, and communication between services is performed via a network and special network queues. This is, of course, neither possible nor desirable for mobile apps, but the basic idea of splitting different features into smaller modular pieces can be adopted, as shown in the following.

The architecture is based on a core containing all the interfaces and API's, but nearly no implementation. All implementations are provided as separated modules. If a module needs an implementation of an interface, it can get the required implementation with the help of *dependency injection*.

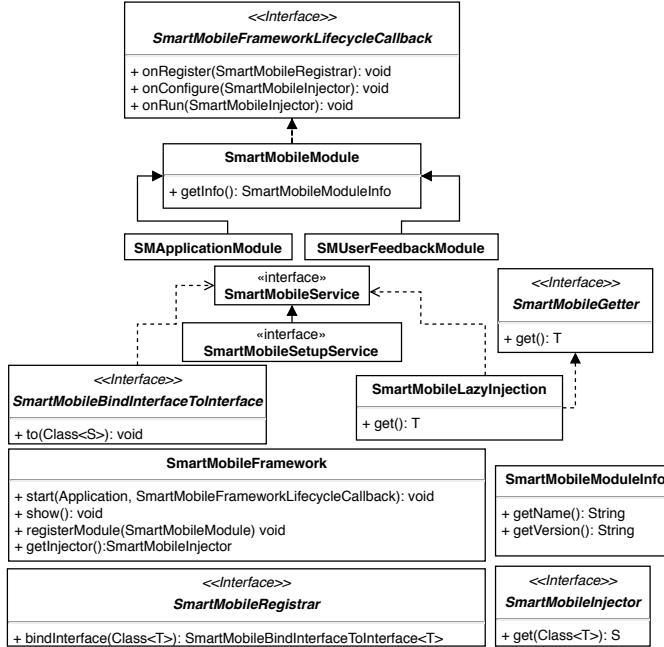
Basically, the complete architecture is based on two main parts:

**The "framework" itself** It contains all interfaces of all the services, that can be injected

**The different "modules"** Every module implements a part of the interfaces defined in the framework

Figure 3 gives an insight into how the framework is structured. This structure makes it very easy to:

- (1) Exchange implementations: Just register another implementation of a module or a service to exchange functionality. For example, the tiles-homescreen-module can easily exchanged with an list-homescreen-module.
- (2) Add functionality: register a new component to the framework without implementing any interface available in the framework. This module can use all other interfaces and their dependent implementation.
- (3) Adapt implementations: This can be done by exchanging modules with an adapted implementation. Still, this solution



**Figure 3: Smart Mobile Framework**

is not optimal, as code reuse must be enforced. It is very important to note that some very module-local adaptations are easy to adapt, but very global changes can be very tricky, as the interfaces should always stay stable and should never be modified for individual customers.

The issue to customize a branding was tackled by the introduction of a special module called 'defaultValues'. This module contains all default texts, icons, settings, and colors of our app. If a module needs a view, it will, e.g., color the view items with the color specified from the default values, or a specified color is used to tint the used icons. This can be done by injecting the assetsService and requesting the specific color:

```
private SmartMobileLazyInjection<SMAssetsService>
    mAssetsService = new SmartMobileLazyInjection<>(
        SMAssetsService.class);
mAssetsService.get().getColorForName("navbar_icon_tint");
```

In order to allow customer-individual changes, an `assetsSetupService` is provided, which allows the customer-specific app to override default values with customer-specific ones. The implementation of these techniques depends on the platform: For Android, values from modules can easily be overwritten with default Android behavior of resources; for iOS a so-called "Bundle" must be provided by the customer app and added to the `assetsSetupService`. To demonstrate how the framework operates, we will take a look at a simple example: Let us assume that we want to add a tile to the main screen. If a user clicks the tile, an alert should display the current time, and if the user accepts the time, this timestamp should be stored securely on the phone. To perform this task, the app has to get an instance of the mainscreen setup service and register a tile to it.

```
private SmartMobileLazyInjection<SMMainScreenSetupService>
    mMainscreenSetupService = new SmartMobileLazyInjection<>(
        SMMainScreenSetupService.class);

mMainscreenSetupService.registerActionForKey("testTile", new
    SmartMobileAction() {
    private SmartMobileLazyInjection<SMAlertViewService>
        malertViewService = new SmartMobileLazyInjection<>(
            SMAlertViewService.class);
    execute() {
        final Date currentDate = new Date();
        malertViewService.showOkAbortDialog("Question", "Do
you accept this time: " + currentDate + "?", new
SMOkAbortCallback() {
            private SmartMobileLazyInjection<SMDatystoreBuilder>
                mDataStoreBuilder = new
SmartMobileLazyInjection<>(SMDatystoreBuilder.class);
            private SMKeyValueDataStore mDataStore =
mDataStoreBuilder.createUserspecificKeyValueStore("test_tile_database_id");
            public onOkPressed() {
                mDataStore.get().set("last_accepted_date",
currentDate);
            }
        }
    }
}
} A tile is referenced only by an id and can be added to a customer specific main screen configuration file. This allows customers to decide individually which tiles they want to see on the main screen; only the functionality must be registered with the shown principle in the framework itself. The action registered for this id is executed as soon as the user clicks on the tile.
```

The smart mobile framework is separated into three different phases: register, configure, and run. Its purpose can best be described by defining a module. A module itself has three different kinds of tasks: It can bind implementations to interfaces (in the so-called register phase of our framework). For example, the main screen module should bind the main screen setup service implementation to the interface:

```
protected void register(SmartMobileRegistrar {
    smartMobileRegistrar.bindInterface(
        MainScreenSetupService.class).to(
        MainScreenSetupServiceImpl.class);
```

After the register phase of every module is finished, it should be possible to inject a suitable implementation for every interface in the framework. In the so-called configure phase, a module should be able to configure other modules; e.g., the example from above should configure the main screen by registering an action for a specific key. The app itself is also a module and can therefore also configure other modules. For example, the `assetsSetupService` can be filled with the customer-specific assets bundle in iOS or the file path to the main screen definition file can be configured for the mainscreen. This is the second task of a module: It can provide

actions to other services in order to provide functionality that can be triggered by a user. Within the configure phase, only setup services can be injected, but no "normal" services. For example, one cannot open the main screen or open an alert from within the configure phase.

```
private final SmartMobileLazyInjection<
    MainScreenSetupService> mMainScreenSetupService = new
        SmartMobileLazyInjection<>(MainScreenSetupService.class
    );

protected void onConfigure(SmartMobileInjector
    smartMobileInjector) {
    mMainScreenSetupService.get().setMainScreenJsonFile("mainscreen.json");
}
```

} After these two phases, the app can start and all normal services can be used. For example, the main screen of the app can be displayed, which now has information about the items it should show, as the configure phase of every module is finished. This constitutes the last task of modules: run the functionality if it is triggered from somewhere.

In summary, the most beneficial changes that could be observed with this new architecture are that now, every module implements a very specific set of services. A module can be considered as standalone library and must only conform to the interfaces, which also allows to use different programming languages for each module. Further, code reuse between the customers is enforced and because of the adaption support any customizations are more efficient to handle. Derived applications are merely correctly puzzled modules. With the framework, it was introduced that customer-specific parts can no longer affect other implementations. We also cleaned up the user interface and made many elements reusable, such as spinners and alerts. But the framework also brings new challenges that need to be considered. For example, we need to take care that the interfaces are kept clean and stable. Further, finding appropriate points where to split functionality into separated modules is not always trivial. If a customer wants changes that influence the architecture and interfaces completely, this can get complex, as a lot of modules must be adapted.

#### 4.4 Feasibility of Further PL Improvements

Since the product line has grown in an evolutionary manner, the ideas of product line development have not been adhered to from the beginning. When it was recognized that the scope was too broad and the framework was introduced, special attention was paid to reuse and high adaptability. A reasonable next step would now be to take the processes and ideas of product line engineering to the next level. To this end, a feasibility study was carried out to see whether further product line engineering practices could also be applied here. In the first step, example scoping was conducted to identify and characterize the different existing variants. Special attention was paid to already existing variation points, since these are now to be organized better in domain engineering. It was revealed that the mass of different variation points has strongly increased and thus it was partly unclear what exactly is covered by domain engineering

and what has to be developed specifically during the course of application engineering. A feature model was developed to provide an overview of the variable features offered by the product line. This is expected to support the requirements engineering process at the application level so that communication about optional and mandatory features can be improved. This will ensure that the requirements for the app will always be complete and valid. At first, no further assets for domain engineering shall be newly developed explicitly, but the framework and the existing components shall be used as a basis of the domain. Since the framework does not yet cover the cross-platform challenge completely, it also needs to be considered how best to deal with this problem. Since cross-platform programming has proved difficult, the possibility remains that the techniques, such as using services, or the interfaces that need to be implemented will be very similar on both platforms. Even if we need to develop the application twice in domain engineering, the application engineering should thus become more efficient and qualitative.

Further, it has to be discussed whether the application engineering can be semi-automated. Therefore, a product configurator could use a consolidated snapshot of a skeleton app of the domain, which could then be customized individually. This would require all variation points to be specified by representative markers, which are instantiated automatically. However, this results in additional effort for maintenance. Nevertheless, in the feasibility study it was noticed that structuring the variability and its organization can yield an advantage for the framework, which is why it should improve the product line in the future.

## 5 CONCLUSION AND OUTLOOK

Based on the challenges of mobile application development, we have seen that this industry has high demand for paradigms supporting development with large-scale structured code reuse. Above all, development for different platforms and various devices and programming languages makes it difficult to introduce structured reuse across platforms. Insiders also faced this problem with its product when several customers requested a similar product. We realized that the approach of cloning and adapting the product became ineffective and error-prone. In addition to the variant drivers inherent in the domain, new drivers also emerged from customer requests. Therefore, a framework was developed that is characterized by a high degree of adaptability, interchangeability, and extensibility. With the help of this framework, new versions can be developed more efficiently and customer requirements can be implemented better. The idea of microservices was used as a guideline to enable individual modules to be integrated or replaced at will in a core. The framework structure was explained and different phases of the framework as well as the tasks of some individual modules were also presented. Above all, code can now be reused across the various customers. The development of the framework was initially detached from the classical approaches of product line engineering. Thus, it was only checked afterwards which additional approaches can be applied to the product line. In this process we realized that a feature model might be helpful to organize the different variants and to create valid configurations.

Since some developers in the industry seem to have similar problems, it might be worth considering further approaches for structured reuse, especially across platforms.

## REFERENCES

- [1] A. Ahmad, Feng, C., Tao, M., Yousif, A., and Ge, S. 2017. Challenges of mobile applications development: Initial results. (2017), 464–469.
- [2] Hamid Bagheri, Joshua Garcia, Alireza Sadeghi, Sam Malek, and Nenad Medvidovic. 2016. Software architectural principles in contemporary mobile software: from conception to practice. *Journal of Systems and Software* 119 (2016), 31 – 44. <https://doi.org/10.1016/j.jss.2016.05.039>
- [3] Jan Bosch. 2005. Software Product Families in Nokia. In *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26–29, 2005, Proceedings* 2–6. [https://doi.org/10.1007/11554844\\_2](https://doi.org/10.1007/11554844_2)
- [4] Josh Dehlinger and Jeremy Dixon. 2011. Mobile application software engineering: Challenges and research directions. 2 (2011), 29–32.
- [5] R. Francese, M. Risi, G. Tortora, and G. Scanniello. 2013. Supporting the development of multi-platform mobile applications. In *2013 15th IEEE International Symposium on Web Systems Evolution (WSE)*. 87–90. <https://doi.org/10.1109/WSE.2013.6642422>
- [6] Nicolas Fußberger, Bo Zhang, and Martin Becker. 2017. A Deep Dive into Android's Variability Realizations. In *Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume A, Sevilla, Spain, September 25–29, 2017*. 69–78. <https://doi.org/10.1145/3106195.3106213>
- [7] Ari Jaaksi. 2002. Developing Mobile Browsers in a Product Line. *IEEE Software* 19, 4 (2002), 73–80. <https://doi.org/10.1109/MS.2002.1020290>
- [8] J. B. Jørgensen, B. Knudsen, L. Sloth, J. R. Vase, and H. B. Christensen. 2016. Variability Handling for Mobile Banking Apps on iOS and Android. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 283–286. <https://doi.org/10.1109/WICSA.2016.29>
- [9] M. E. Joorabchi, Mesbah, A., and Kruchten, P. 2013. Real Challenges in Mobile App Development. (Oktobet 2013), 15–24.
- [10] José A Montenegro, Mónica Pinto, and Lidia Fuentes. 2017. What do software developers need to know to build secure energy-efficient Android applications? *IEEE Access* 6 (2017), 1428–1450.
- [11] Dirk Muthig, Isabel John, Michalis Anastasopoulos, Thomas Forster, Jörg Dörr, and Klaus Schmid. 2004. GoPhone-a software product line in the mobile phone domain. *IESE-Report No 25* (2004), 1–104.
- [12] Dmitry Namiot and Manfred Sneps-Sneppe. 2014. On micro-services architecture. *International Journal of Open Information Technologies* 2, 9 (2014), 24–27.
- [13] Ricardo E. V. De S. Rosa and Vicente Ferreira de Lucena. 2011. Smart composition of reusable software components in mobile application product lines. In *PLEASE@ICSE*.
- [14] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*. 583–590. <https://doi.org/10.1109/ColumbianCC.2015.7333476>

# Static Analysis of Featured Transition Systems

Maurice H. ter Beek\*

ISTI-CNR, Pisa, Italy

maurice.terbeek@isti.cnr.it

Ferruccio Damiani\*

University of Turin, Turin, Italy

ferruccio.damiani@unito.it

Michael Lienhardt\*

ONERA, Palaiseau, France

michael.lienhardt@onera.fr

Franco Mazzanti\*

ISTI-CNR, Pisa, Italy

franco.mazzanti@isti.cnr.it

Luca Paolini\*

University of Turin, Turin, Italy

luca.paolini@unito.it

## ABSTRACT

A Featured Transition System (FTS) is a formal behavioural model for software product lines, which represents the behaviour of all the products of an SPL in a single compact structure by associating transitions with features that condition their existence in products. In general, an FTS may contain featured transitions that are unreachable in any product (so called *dead* transitions) or, on the contrary, mandatorily present in all products for which their source state is reachable (so called *false optional* transitions), as well as states from which only for certain products progress is possible (so called *hidden deadlocks*). In this paper, we provide algorithms to analyse an FTS for such ambiguities and to transform an ambiguous FTS into an unambiguous FTS. The scope of our approach is twofold. First and foremost, an ambiguous model is typically undesired as it gives an unclear idea of the SPL. Second, an unambiguous FTS paves the way for efficient family-based model checking. We apply our approach to illustrative examples from the literature.

## CCS CONCEPTS

- Software and its engineering → Specification languages; Formal methods; Software product lines.

## KEYWORDS

software product lines, formal specification, behavioural model, featured transition systems, static analysis

### ACM Reference Format:

Maurice H. ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini. 2019. Static Analysis of Featured Transition Systems. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3336294.3336295>

## 1 INTRODUCTION

Systems and Software Product Line Engineering (SPL) advocates the reuse of components (systems as well as software) throughout

\* All authors contributed equally to this research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336295>

all phases of product development. Following this paradigm, many businesses no longer develop single products, but rather a family or product line of closely-related, customisable products. This requires identifying the relevant features of the product domain to best exploit their commonality and manage their variability. A feature diagram or feature model defines those combinations of features that constitute valid product configurations [2].

While automated analysis of such structural variability models (e.g., detection of anomalies like dead or false optional features) has a 30-year history [20, 61], that of behavioural variability models has received considerable attention only during the last decade, following the seminal paper by Classen et al. [32]. Given that SPLs often concern massively (re)used and critical software (e.g., in smartphones and the automotive industry) it is important to demonstrate not only their correct configuration, but also their correct behaviour.

A Featured Transition System (FTS) is a formal behavioural model for SPLs, which represents the behaviour of all the products of an SPL in a single compact structure by associating transitions with features that condition their existence in products [30]. Quality assurance by means of model checking or testing is challenging, since ideally it must exploit the compact structure of the FTS to reason on the entire SPL at once. This is called family-based analysis, as opposed to enumerative product-based analysis in which every product is examined individually [60]. During the past decade, FTSs have shown to be amenable to family-based testing and model-checking [10, 11, 28–32, 34, 42–47, 52].

In this paper, we are interested in automated static analysis of FTSs. We want to catch (and offer a means to remove) possible ambiguities in FTSs, mimicking the anomaly detection as typically performed on feature models. In fact, an FTS may contain: *dead* transitions (i.e., featured transitions that are unreachable in any product); *false optional* transitions (i.e., featured transitions that are mandatorily present in all products for which their source state is reachable); and *hidden deadlocks* (i.e., states from which only for certain products progress is possible).

The contribution of this paper is a formalisation of ambiguities in FTSs and algorithms to analyse an FTS for such ambiguities and to transform an ambiguous FTS into an unambiguous one, as well as proofs of their correctness. The scope is twofold. First, in analogy with the anomalies in feature models, an ambiguous FTS is often undesired since it gives an unclear idea of the SPL. Second, an unambiguous FTS paves the way for efficient family-based model checking, because it has specific characteristics that enable model checking of properties expressed in a rich, action-based and variability-aware fragment of the well-known CTL logic directly on the FTS such that validity is preserved in all products.

We apply our approach to illustrative examples from the literature.

*Related work.* Static analysis of FTSs mimics the automated analysis of feature models [20, 61], by defining behavioural counterparts of dead and false optional features. It is related to static (program) analysis [24, 58], which includes the detection of bugs in the code (like using a variable before its initialisation) but also the identification of code that is redundant or unreachable. In [52], conventional static analysis techniques are applied to SPLs represented in the form of object-oriented programs with feature modules. The aim is to find irrelevant features for a specific test in order to use this information to reduce the effort in testing an SPL by limiting the number of SPL programs to examine to those with relevant features. In [23], several well-known static analysis techniques are lifted to SPLs without the exponential blowup caused by generating and analysing all products individually, by converting such analyses to feature-sensitive analyses that operate on the entire SPL in one single pass. In [51], static type checking is extended from single programs to an entire SPL by extending the type system of a subset of Java with feature annotations. This guarantees that whenever the SPL is well-typed, then all possible program variants are well-typed as well, without the need for generating and compiling them first. An encompassing overview of (static) analysis strategies for SPLs can be found in [60] and a recent empirical study on applying variability-aware static analysis techniques to real-world configurable systems is presented in [59].

*Outline.* After some background (in Sect. 2), we define ambiguities in FTSs (in Sect. 3); provide criteria that enable to identify them by static analysis (in Sect. 4); present a static analysis algorithm to detect ambiguities in FTSs (in Sect. 5); illustrate the application of the algorithm on FTSs from the literature (in Sect. 6); discuss the usefulness of removing ambiguities from FTSs (in Sect. 7) and scalability issues of our approach (in Sect. 8); after which we conclude by briefly outlining some planned future work (in Sect. 9).

## 2 BACKGROUND

In this section, we provide some background needed for the sequel. We recall LTSs as the underlying behavioural structure of FTSs.

*Definition 2.1 (LTS).* A *Labelled Transition System* (LTS) is a quadruple  $(S, \Sigma, s_0, \delta)$ , where  $S$  is a finite (non-empty) set of states,  $\Sigma$  is a set of actions,  $s_0 \in S$  is an initial state, and  $\delta \subseteq S \times \Sigma \times S$  is a transition relation.

We call  $(s, a, s') \in \delta$  an  $a$ -labelled transition (from source state  $s$  to target state  $s'$ ) and we may also write it as  $s \xrightarrow{a} s'$ .

We recall classical notions for LTSs that we will use in the paper.

*Definition 2.2 (reachability).* Let  $\mathcal{L} = (S, \Sigma, s_0, \delta)$  be an LTS. A sequence  $p = s_0 t_1 s_1 t_2 s_2 \dots$  is a *path* of  $\mathcal{L}$  if  $t_i = (s_{i-1}, a_i, s_i) \in \delta$  for all  $i > 0$ ;  $p$  is said to *visit* states  $s_0, s_1, \dots$  and transitions  $t_1, t_2, \dots$  and we denote its  $i$ th state by  $p(i)$  and its  $i$ th transition by  $p\{i\}$ .

A state  $s \in S$  is *reachable* (via  $p$ ) in  $\mathcal{L}$  if there exists a path  $p$  that visits it, i.e.,  $p(i) = s$  for some  $i \geq 0$ ;  $s$  is a *deadlock* if it has no outgoing transitions, i.e.,  $\nexists (s, a, s') \in \delta$ , for all  $a \in \Sigma$  and  $s' \in S$ .

A transition  $t = (s, a, s') \in \delta$  is *reachable* (via  $p$ ) in  $\mathcal{L}$  if there exists a path  $p$  that visits it, i.e.,  $p\{i\} = t$ , for some  $i > 0$ .

FTSs were introduced in [30, 32] to concisely model the behaviour of all the products of an SPL in one transition system by annotating transitions with conditions expressing their existence in products. Let  $\mathbb{B} = \{\top, \perp\}$  denote the Boolean constants true ( $\top$ ) and false ( $\perp$ ), and let  $\mathbb{B}(F)$  denote the set of Boolean expressions over a set of features  $F$  (i.e., using features as propositional variables). We do not formalise a language for Boolean expressions in order to allow the inclusion of all possible propositional connectives and, in particular, we include the constants from  $\mathbb{B}$ . The elements of  $\mathbb{B}(F)$  are also called *feature expressions*. An FTS is an LTS equipped with a feature model and a function that labels each transition with a feature expression. In the following definition, the feature model is represented by the set of its (product) configurations, where each configuration is represented by a Boolean assignment to the features (i.e., selected =  $\top$  and unselected =  $\perp$ ).

*Definition 2.3 (FTS).* A *Featured Transition System* (FTS) is a sextuple  $(S, \Sigma, s_0, \delta, F, \Lambda)$ , where  $S$  is a finite (non-empty) set of states,  $\Sigma$  is a set of actions,  $s_0 \in S$  is the initial state,  $\delta \subseteq S \times \Sigma \times \mathbb{B}(F) \times S$  is a transition relation,  $F$  is a set of features, and  $\Lambda \subseteq \{\lambda : F \rightarrow \mathbb{B}\}$  is a set of (product) configurations. Without loss of generality, we assume that whenever  $(s, a, \phi, s'), (s, a, \phi', s') \in \delta$ , then  $\phi = \phi'$  (*transition injectivity*).

We call  $(s, a, \phi, s') \in \delta$ , for some feature expression  $\phi \in \mathbb{B}(F)$ , a *featured transition* (labelled with  $a$  and limited to configurations satisfying  $\phi$ ) and we call  $(s, a, \top, s') \in \delta$  a *must transition*. We may also write featured transitions as  $s \xrightarrow{a \mid \phi} s'$ .

The notions from Definition 2.2 (path, reachability, deadlock) are carried over to FTSs by ignoring the feature expressions. The transition injectivity in Definition 2.3, guaranteeing that a transition identifies a unique feature expression (as in [21, 62]), turns out to be useful for some of the technical results in this paper. Moreover, we know from [30] (Theorem 8) that restricting feature expressions to singleton features does not affect expressiveness.

A configuration  $\lambda \in \Lambda$  satisfies a feature expression  $\phi \in \mathbb{B}(F)$ , denoted by  $\lambda \models \phi$ , whenever the interpretation of  $\phi$  via  $\lambda$  is true, i.e., if the result of substituting the value of the features occurring as variables in  $\phi$  according to  $\lambda$  is  $\top$ . By definition,  $\lambda \models \top$ .

*Definition 2.4 (product).* Let  $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$  be an FTS. The LTS specified by a particular configuration  $\lambda \in \Lambda$ , denoted by  $\mathcal{F}|_\lambda$ , is called a *product* of  $\mathcal{F}$ . It is obtained from  $\mathcal{F}$  by first removing all featured transitions whose feature expressions are not satisfied by  $\lambda$  (resulting in the LTS  $(S, \Sigma, s_0, \delta')$ , with  $\delta' = \{(s, a, s') \mid (s, a, \phi, s') \in \delta \text{ and } \lambda \models \phi\}$ ), and then removing all unreachable states and their outgoing transitions. The set of products of  $\mathcal{F}$  is denoted by  $\text{lts}(\mathcal{F})$ .

Note that, by construction: (i) each product does not contain unreachable states or transitions; (ii) each must transition of the FTS is maintained in the products in which it is reachable; and (iii) each product does not contain states or actions that were not originally present in the FTS.

Let  $\text{fm}_{\mathcal{F}}$  denote the *feature model expression* of  $\mathcal{F}$ , i.e., a feature expression that represents  $\Lambda$  (like, e.g., the formula, in conjunctive normal form,  $\bigvee_{\lambda \in \Lambda} (\bigwedge_{f \in F} (\{f \mid \lambda(f) = \top\} \cup \{\neg f \mid \lambda(f) = \perp\}))$ ). We may write  $\text{fm}$  instead of  $\text{fm}_{\mathcal{F}}$  if no confusion can arise.

*Example 2.5.* In Fig. 1, we depict an FTS  $\mathcal{F}$ , modelling the behaviour of a product line of coffee machines, adapted from [10].

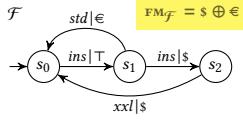
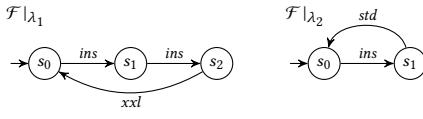


Figure 1: FTS of a product line of coffee machines

Figure 2: LTSSs of products  $\lambda_1$  and  $\lambda_2$  of the FTS of Fig. 1

This FTS has transitions labelled with features  $\$$  and  $\epsilon$ , representing products for either the American or the European market, respectively, and a must transition that must be present in every product. Its feature model can thus be represented by the feature expression  $\text{FM}_{\mathcal{F}} = \$ \oplus \epsilon$ , where  $\oplus$  denotes the *exclusive disjunction* operation. Hence the product configurations of  $\mathcal{F}$  are  $\Lambda = \{\lambda_1, \lambda_2\}$ , where  $\lambda_1(\$) = \top, \lambda_1(\epsilon) = \perp, \lambda_2(\$) = \perp$ , and  $\lambda_2(\epsilon) = \top$ .

The FTS has actions to insert coins (*ins*) and to pour standard (*std*) or extra large (*xxl*) coffee. Extra large coffee is exclusively available for the American market (for two dollars), while standard coffee is exclusively available for the European market (for one euro). The LTSSs  $\mathcal{F}|_{\lambda_1}$  and  $\mathcal{F}|_{\lambda_2}$ , depicted in Fig. 2, model the behaviour of the two products of  $\mathcal{F}$ : configuration  $\lambda_1$  for the American market and configuration  $\lambda_2$  for the European market.

### 3 AMBIGUITIES IN FTSs

Some of the better known analysis operations that are typically performed as part of the automated analysis of feature models concern the detection of anomalies (cf., e.g., [20, 61]). These anomalies reflect ambiguous or even contradictory information. Examples include so-called dead and false optional features. A feature is *dead* if it is not contained in any of the products of the SPL, typically due to an incorrect use of cross-tree constraints, whereas a feature is *false optional* if it is contained in all the products of the SPL even though it is not a designated mandatory feature.

In this section, we capture equivalent notions in a behavioural setting, by adapting them to (featured) transitions of an FTS: we define ambiguous FTSs (Sect. 3.1) and show how to transform an ambiguous FTS into an unambiguous one (Sect. 3.2).

#### 3.1 Behavioural Ambiguities

**Definition 3.1 (dead transition).** We define a transition (of an FTS) to be *dead* if it is not reachable in any of the FTS's products.

**Definition 3.2 (false optional transition).** A featured transition (of an FTS) is *false optional* if: (i) it is not annotated with the feature expression  $\top$  and (ii) it is present in all the FTS's products in which its source state is reachable.

An important safety property of reactive systems concerns *deadlock freedom*, i.e., the system should not reach a state in which

no further action is possible, thus guaranteeing progress or liveness [1, 56]. In case of configurable reactive systems, like SPLs, this notion can be extended to guaranteeing liveness for each system variant (product). Recall from Sect. 2 that a state of an FTS is said to be a deadlock if it has no outgoing transitions and from Definition 2.4 that all states of a product (LTS) of an FTS are reachable.

**Definition 3.3 (hidden-deadlock state).** We define a state (of an FTS) to be a *hidden deadlock* if: (i) it is not a deadlock in the FTS and (ii) it is a deadlock in one or more of the FTS' products (LTSs).

**Definition 3.4 (ambiguous FTS).** An FTS is said to be *ambiguous* if any of its states is a hidden deadlock or if any of its transitions is dead or false optional.

**Example 3.5.** In Fig. 3(left), we depict an FTS  $\mathcal{F}$  with features  $f_1$  and  $f_2$  and feature model  $\text{FM} = f_1 \oplus f_2$ .

The LTSSs  $\mathcal{F}|_{\lambda_1}$  and  $\mathcal{F}|_{\lambda_2}$ , depicted in Fig. 4(left and middle), model the behaviour of its two valid product configurations:  $\lambda_1 = \{f_1\}$  and  $\lambda_2 = \{f_2\}$ . We immediately see that featured transition  $s_2 \xrightarrow{a \mid f_2} s_2$  is dead, featured transition  $s_1 \xrightarrow{a \mid f_1} s_2$  is false optional, and state  $s_2$  is a hidden deadlock. Hence  $\mathcal{F}$  is an ambiguous FTS.

#### 3.2 Removing Ambiguities

Any ambiguous FTS can be straightforwardly turned into an unambiguous FTS by the following transformation:

- (1) remove its dead transitions;
- (2) turn its false optional transitions into must transitions; and
- (3) make its hidden deadlocks explicit by adding to  $Q$  a distinguished deadlock state  $s_{\dagger} \notin Q$  and, for each hidden deadlock state  $s$ , adding a new transition (which we call a *deadlock transition*) with  $s$  as source,  $s_{\dagger}$  as target, and labelled by a distinguished action  $\dagger \notin \Sigma$  and by a feature expression that negates the disjunction of the feature expressions of all its source state's outgoing transitions.

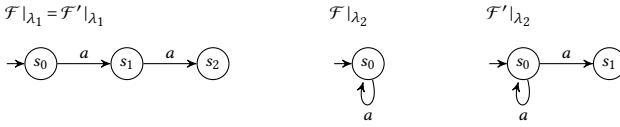
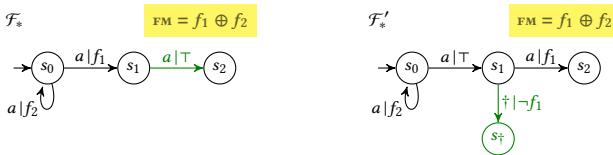
Note that step (3) needs to be performed only for those hidden deadlock states that have not yet become explicit deadlock states upon the removal of dead featured transitions in step (1).

**Example 3.6.** In Fig. 5(left), we depict an unambiguous FTS  $\mathcal{F}_*$  that was obtained by transforming the ambiguous FTS  $\mathcal{F}$  of Fig. 3. We removed dead featured transition  $s_2 \xrightarrow{a \mid f_2} s_2$  and false optional featured transition  $s_1 \xrightarrow{a \mid f_1} s_2$  was turned into must transition  $s_1 \xrightarrow{a \mid \top} s_2$ . Note that there was no need to add an explicit deadlock transition from the hidden deadlock state  $s_2$  to a newly added explicit deadlock state, since  $s_2$  became an explicit deadlock state upon removing the dead featured transition  $s_2 \xrightarrow{a \mid f_2} s_2$ .

Now consider the ambiguous FTS  $\mathcal{F}'$  depicted in Fig. 3(right) with features  $f_1$  and  $f_2$  and feature model  $\text{FM} = f_1 \oplus f_2$ . The LTSSs  $\mathcal{F}'|_{\lambda_1}$  and  $\mathcal{F}'|_{\lambda_2}$ , depicted in Fig. 4(left and right), model the behaviour of its two valid product configurations:  $\lambda_1 = \{f_1\}$  and  $\lambda_2 = \{f_2\}$ . Similar to Example 3.5, featured transition  $s_2 \xrightarrow{a \mid f_2} s_2$  is dead. However, featured transition  $s_1 \xrightarrow{a \mid f_1} s_2$  is no longer false optional, since it is indeed not present in  $\mathcal{F}'|_{\lambda_2}$  even though its source state  $s_1$  is reachable in that LTS. Moreover, not only state  $s_2$  is a hidden deadlock (for the same reason as before) but so is state  $s_1$ , since it is a deadlock in  $\mathcal{F}'|_{\lambda_2}$ . Hence also  $\mathcal{F}'$  is ambiguous.



Figure 3: Ambiguous FTSs

Figure 4: LTSs of products  $\lambda_1$  and  $\lambda_2$  of the FTSs of Fig. 3

In Fig. 5(right), we depict an unambiguous FTS  $\mathcal{F}'^*$  that was obtained by transforming the ambiguous FTS  $\mathcal{F}'$  of Fig. 3 as follows. We removed the dead featured transition  $s_2 \xrightarrow{a|f_2} s_2$  and we added the explicit deadlock transition  $s_1 \xrightarrow{\dagger \mid \neg f_1} s_{\dagger}$  from the hidden deadlock state  $s_1$  to the newly added explicit deadlock state  $s_{\dagger}$ . Note that in this case, without adding this explicit deadlock transition, state  $s_1$  would remain a hidden deadlock state in  $\mathcal{F}'^*$ .

## 4 CRITERIA FOR AMBIGUITIES

The ambiguities in FTSs can be characterised by criteria that enable the definition of the static analysis algorithm given in Sect. 5.

Our goal is to spot ambiguities by exploring once the whole FTS in order to infer properties holding for all its products. It is worth noting that the reachability of a state in an FTS (via a path) is not a sufficient condition to ensure that there exist a product including such a state (because the conjunction of the feature expressions of the transitions in each path reaching the state may be false for all configurations). Anyway, the set of configurations of the FTS contains sufficient information to identify its products. In order to circumvent the generation of all products and analyse them one by one, we use Boolean formulas suitable for SAT solver processing.

**Definition 4.1 (FTS notations).** Let  $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$  be an FTS.

- (1) For a transition  $t = (s, a, \phi, s') \in \delta$ , we denote its source state  $s$  by  $t.\text{SOURCE}$ , its target state  $s'$  by  $t.\text{TARGET}$ , and its feature expression  $\phi$  by  $t.\text{BX}$ .
- (2) We write  $\text{PATHS}(s)$  to denote the set of all (finite) paths (starting from the initial state  $s_0$  and) ending in state  $s \in S$ , and we write  $\text{PATHS}(t)$  to denote the set of all (finite) paths (starting from the initial state  $s_0$  and) ending with transition  $t \in \delta$ .
- (3) For a path  $p$ , we define its *path expression*, denoted by  $\text{BX}_p$ , as the formula  $\wedge_{t \in p} (t.\text{BX})$ , i.e., the conjunction of all formulas labelling the transitions visited by  $p$ .

Finiteness of an FTS does not imply that the set  $\text{PATHS}(s)$  is finite. For instance, the FTS  $\mathcal{F}$  of Fig. 1 is such that  $\text{PATHS}(s_0)$  contains an infinite number of paths of the form  $s_0 t_1 s_1 t_2 s_0 \dots s_0 t_1 s_1 t_2 s_0$ , where  $t_1 = (s_0, \text{ins}|\top, s_1)$  and  $t_2 = (s_1, \text{std}|\ell, s_0)$ .

**Definition 4.2 (cycle-free path).** Let  $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$  be an FTS. A path is *cycle-free* whenever it visits each state at most once. Let  $\text{cfPATHS}(s)$  denote the set of all cycle-free paths ending in state  $s \in S$ . Let  $\text{cfPATHS}(t)$  denote the set of all cycle-free paths ending with transition  $t \in \delta$ .

Since a cycle-free path visits each transition at most once, finiteness of an FTS ensures that  $\text{cfPATHS}(t)$  is finite. A cycle-free path  $p$  can be seen as the canonical representative of the equivalence class of the paths that become  $p$  when removing the cycles.

**LEMMA 4.3.** Let  $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$  be an FTS and let  $s \in S$ . If  $p \in \text{PATHS}(s)$ , then the path  $q \in \text{cfPATHS}(s)$  obtained from  $p$  by removing its cycles is such that  $\text{BX}_p \Rightarrow \text{BX}_q$ .

**PROOF.** By construction, for all transitions  $t$  it holds that  $t \in q$  implies  $t \in p$ . Thus, by Definition 4.1,  $\text{BX}_p \Rightarrow \text{BX}_q$  holds.  $\square$

## 4.1 Dead Transitions Criterion

The next lemma formalises Definition 3.1 in terms of an FTS' paths, explicating that to decide whether a transition is dead it suffices to inspect the path expressions of all paths ending with that transition. Then, Lemma 4.5 states that it suffices to consider cycle-free paths.

**LEMMA 4.4.** Let  $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$  be an FTS, let  $t \in \delta$ , and let  $\text{PATHS}(t) \neq \emptyset$ . The transition  $t$  is dead if and only if  $\text{FM}_{\mathcal{F}} \wedge \text{BX}_p$  is not satisfiable, for all  $p \in \text{PATHS}(t)$ .

**PROOF.** ( $\Rightarrow$ ) We prove the contrapositive. Let  $p^* \in \text{PATHS}(t)$  be the path  $s_0 t_1 s_1 \dots s_{n-1} t_n s_n$  such that  $\text{FM} \wedge \text{BX}_{p^*}$  is satisfiable. A formula is satisfiable whenever it can be made true by assigning appropriate logical values to its variables; namely, there exists a configuration  $\lambda^*$  that assigns logical values to features such that  $\lambda^* \models \text{FM} \wedge \text{BX}_{p^*}$ ; namely, both  $\lambda^* \models \text{FM}$  and  $\lambda^* \models \text{BX}_{p^*}$ . But  $\lambda^* \models \text{FM}$  simply means that  $\lambda^* \in \Lambda$ . Moreover,  $\lambda^* \models \text{BX}_{p^*}$  means that, for all  $1 \leq i \leq n$ , if  $t_i = (s_{i-1}, a_i, \phi_i, s_i)$ , then  $\lambda^*(\phi_i) = \top$ ; this is trivial in case  $\phi_i = \top$ . Concluding,  $p^*$  identifies a path reaching  $t$  in the product  $\mathcal{F}|_{\lambda^*}$  defined by  $\lambda^*$ , so  $t$  is not dead and the proof is done.

( $\Leftarrow$ ) We prove the contrapositive. Let  $t = (s, a, \phi, s')$  and assume that  $t$  is not dead, i.e.,  $t' = (s, a, s')$  is reachable in a product  $\mathcal{F}|_{\lambda^*}$  defined by the configuration  $\lambda^* \in \Lambda$ . Namely,  $t'$  is visited by a path  $p = s_0 t'_1 s_1 t'_2 s_2 \dots s_{n-1} t'_n s_n$  of  $\mathcal{F}|_{\lambda^*}$  such that  $s = s_{n-1}$ ,  $s' = s_n$ , and  $t' = t'_n$ , for some  $n > 0$ . Then, the transition injectivity (cf. Definition 2.3) guarantees there exists a unique list of transitions  $t_1, \dots, t_n \in \delta$  such that  $t_n = t$  and, for all  $1 \leq i \leq n$ ,  $t_i = (s_{i-1}, a_i, \phi_i, s_i)$  for some  $\phi_i$  such that  $\lambda^*(\phi_i) = \top$ . Thus the path  $s_0 t_1 s_1 \dots s_{n-1} t_n s_n$  is included in  $\text{PATHS}(t)$  by Definition 4.2 and  $\lambda^* \models \text{BX}_p$ . As  $\lambda^* \models \text{FM}$ , we conclude that  $\text{FM} \wedge \text{BX}_p$  is satisfiable.  $\square$

**LEMMA 4.5.**  $\text{FM} \wedge \text{BX}_p$  is not satisfiable, for all  $p \in \text{PATHS}(t)$ , if and only if  $\text{FM} \wedge \text{BX}_q$  is not satisfiable, for all  $q \in \text{cfPATHS}(t)$ .

**PROOF.** Direction ( $\Rightarrow$ ) is trivial, because  $\text{cfPATHS}(t) \subseteq \text{PATHS}(t)$ . Direction ( $\Leftarrow$ ) follows by Lemma 4.3, we prove the contrapositive. If there is a path  $p \in \text{PATHS}(t)$  such that  $\text{FM} \wedge \text{BX}_p$  is satisfiable, then we can find a path  $q \in \text{cfPATHS}(t)$  such that  $\text{FM} \wedge \text{BX}_q$  is satisfiable.  $\square$

The next theorem, which follows by Lemmas 4.4 and 4.5, provides an equivalent definition that can be used as an effective criterion to decide whether a transition is dead (cf. Definition 3.1).

**THEOREM 4.6 (DEAD TRANSITIONS CRITERION).**  
Let  $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$  be an FTS, let  $t \in \delta$ , and let  $cfPATHS(t) \neq \emptyset$ . The transition  $t$  is dead if and only if  $FM_{\mathcal{F}} \wedge BX_q$  is not satisfiable, for all  $q \in cfPATHS(t)$ .

## 4.2 False Optional Transitions Criterion

The next lemma formalises Definition 3.2 in terms of an FTS' paths, explicating that to decide whether a transition is false optional it suffices to inspect the path expressions of all paths ending in the source state of that transition. Then, Lemma 4.8 states that it suffices to consider cycle-free paths.

**LEMMA 4.7.** Let  $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$  be an FTS and let  $t \in \delta$  be such that  $t.BX \neq \top$ . The transition  $t$  is false optional if and only if, for all  $p \in PATHS(t.SOURCE)$ ,  $FM_{\mathcal{F}} \Rightarrow (BX_p \Rightarrow t.BX)$  is a tautology.

**PROOF.** ( $\Rightarrow$ ) We recall that  $t$  is false optional whenever for all  $\lambda \in \Lambda$ , for all  $p \in PATHS(t.SOURCE)$ ,  $\lambda \models BX_p$  implies  $\lambda \models t.BX$ . We assume that  $t$  is false optional and prove that  $FM \Rightarrow (BX_p \Rightarrow t.BX)$  is a tautology, for all  $p \in PATHS(t.SOURCE)$ . Let  $\lambda^*$  be an assignment of logical values to all features. We consider two sub-cases. First, assume that  $\lambda^* \not\models FM \wedge BX_p$ . Then immediately  $\lambda^* \models FM \Rightarrow (BX_p \Rightarrow t.BX)$ . Second, assume that  $\lambda^* \models FM \wedge BX_p$ . Clearly  $\lambda^* \in \Lambda$ , because  $\lambda^* \models FM$ . Moreover,  $\lambda^* \models t.BX$  because  $\lambda^* \models BX_p$  and  $t$  is false optional. Therefore, also in this case we conclude  $\lambda^* \models FM \Rightarrow (BX_p \Rightarrow t.BX)$ . Summarising,  $FM \Rightarrow (BX_p \Rightarrow t.BX)$  is a tautology.

( $\Leftarrow$ ) Assume that  $FM \Rightarrow (BX_p \Rightarrow t.BX)$  is a tautology, for all  $p \in PATHS(t.SOURCE)$ . If, for all  $p \in PATHS(t.SOURCE)$ , for all  $\lambda^* \in \Lambda$ ,  $\lambda^* \not\models BX_p$ , then  $t$  is trivially false optional. Hence, we assume that  $\lambda^* \in \Lambda$  is such that  $\lambda^* \models BX_{p^*}$  for a  $p^* \in PATHS(t.SOURCE)$  and we aim to prove that  $\lambda^* \models t.BX$ . Clearly,  $\lambda^* \models FM$  by Definition 4.1. Therefore  $\lambda^* \models FM \Rightarrow (BX_p \Rightarrow t.BX)$  implies  $\lambda^* \models t.BX$ , which completes the proof.  $\square$

**LEMMA 4.8.**  $FM \Rightarrow (BX_p \Rightarrow t.BX)$  is a tautology, for all  $p \in PATHS(t.SOURCE)$ , if and only if  $FM \Rightarrow (BX_q \Rightarrow t.BX)$  is a tautology, for all  $q \in cfPATHS(t.SOURCE)$ .

**PROOF.** Direction ( $\Rightarrow$ ) is trivial, because  $cfPATHS(s) \subseteq PATHS(s)$ . Direction ( $\Leftarrow$ ) is proved by contraposition. Let  $\lambda^*$  be an assignment of logical values to all features such that  $\lambda^* \not\models FM \Rightarrow (BX_p \Rightarrow t.BX)$ , for some  $p \in PATHS(t.SOURCE)$ . This means that  $\lambda^* \models FM \wedge BX_p$  and  $\lambda^* \not\models t.BX$ . By Lemma 4.3, there is a path  $q \in cfPATHS(t)$  such that  $\lambda^* \models FM \wedge BX_q$  but, still,  $\lambda^* \not\models t.BX$ . This proves the logical equivalence.  $\square$

The next theorem, which follows by Lemmas 4.7 and 4.8, provides an equivalent definition that can be used as an effective criterion to decide whether a transition is false optional (cf. Definition 3.2).

**THEOREM 4.9 (FALSE OPTIONAL TRANSITIONS CRITERION).**  
Let  $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$  be an FTS and let  $t \in \delta$  be such that  $t.BX \neq \top$ . The transition  $t$  is false optional if and only if, for all  $p \in cfPATHS(t.SOURCE)$ ,  $FM_{\mathcal{F}} \Rightarrow (BX_p \Rightarrow t.BX)$  is a tautology.

## 4.3 Hidden Deadlock States Criterion

The next lemma formalises Definition 3.3 in terms of an FTS' paths, explicating that to decide whether a state is a hidden deadlock it suffices to inspect the path expressions of all paths ending with a transition having such state as its source state. Then, Lemma 4.11 states that it suffices to consider cycle-free paths.

**LEMMA 4.10.** Let  $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$  be an FTS, let  $s \in S$ , and let  $s.EXIT\_TRS$  denote the set of transitions starting from  $s$ . The state  $s$  is **not** a hidden deadlock if and only if, either  $s.EXIT\_TRS = \emptyset$  or for all  $p \in PATHS(s)$ ,  $FM_{\mathcal{F}} \Rightarrow (BX_p \Rightarrow (\bigvee_{t \in s.EXIT\_TRS} t.BX))$  is a tautology.

**PROOF.** ( $\Rightarrow$ ) A state  $s$  is not a hidden deadlock whenever, either  $s.EXIT\_TRS = \emptyset$  or for all configurations  $\lambda \in \Lambda$  and for all paths  $p \in PATHS(s)$ ,  $\lambda \models BX_p$  implies that there exists a transition  $t \in s.EXIT\_TRS$  such that  $\lambda \models t.BX$ . Assume that  $s$  is not a hidden deadlock. If  $s.EXIT\_TRS = \emptyset$ , then the proof is immediate; hence we assume that for all  $\lambda \in \Lambda$  and for all  $p \in PATHS(s)$ ,  $\lambda \models BX_p$  implies  $\lambda \models (\bigvee_{t \in s.EXIT\_TRS} t.BX)$  and we aim to prove that  $FM \Rightarrow (BX_p \Rightarrow \bigvee_{t \in s.EXIT\_TRS} t.BX)$  is a tautology. The proof is straightforward, because for all  $\lambda \in \Lambda$  we know that  $\lambda \models FM$ .

( $\Leftarrow$ ) If  $s$  is such that  $s.EXIT\_TRS = \emptyset$ , then the proof is immediate. Let  $p \in PATHS(s)$  be such that  $FM \Rightarrow (BX_p \Rightarrow (\bigvee_{t \in s.EXIT\_TRS} t.BX))$  is a tautology. This means that for all assignments  $\lambda$  of logical values to features, the following holds: if  $\lambda \models FM$  and  $\lambda \models BX_p$ , then  $\lambda \models (\bigvee_{t \in s.EXIT\_TRS} t.BX)$ . Clearly,  $\lambda \models FM$  implies  $\lambda \in \Lambda$ , so the proof is straightforward.  $\square$

**LEMMA 4.11.**  $FM \Rightarrow (BX_p \Rightarrow (\bigvee_{t \in s.EXIT\_TRS} t.BX))$  is a tautology, for all  $p \in PATHS(s)$ , if and only if  $FM \Rightarrow (BX_q \Rightarrow (\bigvee_{t \in s.EXIT\_TRS} t.BX))$  is a tautology, for all  $q \in cfPATHS(s)$ .

**PROOF.** Direction ( $\Rightarrow$ ) is trivial, because  $cfPATHS(s) \subseteq PATHS(s)$ . Direction ( $\Leftarrow$ ) is proved by contraposition. Let  $\lambda^*$  be an assignment of logical values to all features such that  $\lambda^* \not\models FM \Rightarrow (BX_p \Rightarrow (\bigvee_{t \in s.EXIT\_TRS} t.BX))$ , for some path  $p \in PATHS(s)$ . This means that  $\lambda^* \models FM \wedge BX_p$  and  $\lambda^* \not\models (\bigvee_{t \in s.EXIT\_TRS} t.BX)$ . By Lemma 4.3, there is a path  $q \in cfPATHS(s)$  such that  $\lambda^* \models FM \wedge BX_q$  but, still,  $\lambda^* \not\models (\bigvee_{t \in s.EXIT\_TRS} t.BX)$ . This proves the logical equivalence.  $\square$

The next theorem, which follows by Lemmas 4.10 and 4.11, provides an equivalent definition that can be used as an effective criterion to decide whether a state is a hidden deadlock (cf. Definition 3.3).

**THEOREM 4.12 (HIDDEN DEADLOCK STATES CRITERION).**  
Let  $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$  be an FTS, let  $s \in S$ , and let  $s.EXIT\_TRS$  denote the set of transitions starting from  $s$ . The state  $s$  is a hidden deadlock if and only if both  $s.EXIT\_TRS \neq \emptyset$  and there exists a path  $p \in cfPATHS(s)$  such that  $FM_{\mathcal{F}} \Rightarrow (BX_p \Rightarrow (\bigvee_{t \in s.EXIT\_TRS} t.BX))$  is **not** a tautology.

## 4.4 SAT Solving

All criteria presented in this section are defined as deciding for a given Boolean formula if it is a tautology or not satisfiable. Checking these criteria are thus variations of the SAT problem [33], where instead of finding if a Boolean formula has a solution, we want to find if it has no solution (e.g., a formula  $\phi$  is a tautology iff  $\neg\phi$

has no solution). We can thus remark that these problems are co-NP-complete. Additionally, SAT solvers can be used to solve these problems. SAT solving is an active field of research [4, 22, 48, 50] and tools exist that compute, more or less efficiently, a solution for an input formula, or fail if the formula is not satisfiable. Hence, by feeding the formula of a criterion to a SAT solver and checking if it fails, we can conclude if the criterion is validated or not. In our implementation, we used the Z3 SMT solver [40, 57] (that includes a SAT solver) developed by Microsoft Research and freely available under the MIT license.

## 5 STATIC ANALYSIS ALGORITHM

In this section, we present an algorithm that visits all cycle-free paths (starting from the initial state) of an FTS in a depth-first order. We describe the algorithm via python pseudocode that identifies the bodies of functions, selection, and iteration operators, by means of a suitable indentation, in place of the more common C-like curly brackets (cf. Listings 1 and 2). The algorithm assumes that as input is provided a suitable representation of an FTS. Then in a *unique* FTS traversal it is able to identify all ambiguities: hidden deadlock states (cf. Definition 3.3) and dead and false optional transitions (cf. Definitions 3.1 and 3.2). We remark that our solution rests on the use of a suitable SAT solver.

**Listing 1: Ambiguities discovery algorithm**

```

1  for t in fts.trs: // initialise transition deadness & optionality
2      t.dead ← true
3      if (t.bx ≠ true): t.false_opt = true
4      else: t.false_opt = false
5  for s in fts.states: s.live ← true // initialise state liveness
6  path_discover(fts.initial, true)
7  for s in fts.states:           // set state hidden deadlockness
8      s.hDead ← (s.exit_trs ≠ ∅ & not (s.live))

```

**Listing 2: Depth-first visit procedure**

```

1  def path_discover(s, bxp):
2      s.visited ← true
3      if(s.live == true & (s.exit_trs ≠ ∅)):
4          s.live ← ((fts.fm ⇒ (bxp ⇒ (∨t ∈ s.exit_trs t.bx))) is tautology)
5      for t ∈ s.exit_trs:
6          if(t.false_opt == true):
7              t.false_opt ← ((fts.fm ⇒ (bxp ⇒ t.bx)) is tautology)
8          ext_bxp ← bxp & t.bx
9          validp ← ((fts.fm & ext_bxp) is satisfiable)
10         if(t.dead == true): t.dead ← not(validp)
11         if(not t.exit_state.visited) & validp:
12             path_discover(t.exit_state, ext_bxp)
13         s.visited ← false

```

First of all, we introduce the structure of global variables. The considered FTS is stored in the global variable `fts`. It has four (relevant) fields:

- (1) `trs` is the set of all the transitions in the FTS;
- (2) `states` is the set of all states in the FTS;
- (3) `initial` is the initial state of the FTS;
- (4) `fm` is the Boolean expression representing the feature model of the FTS.

The structure of states has four fields as well:

- (1) `visited` is a Boolean value indicating if the state is included in the path currently visited;
- (2) `exit_trs` is the set of transitions exiting from this state;

- (3) `live` is a Boolean flag that stores the discovery of products with states (having at least one outgoing transition) that are deadlocks;
- (4) `hdead` is a Boolean flag meant to mark the hidden deadlock status of states.

The data structure of transitions is crucial:

- (1) `bx` is a Boolean expression formalising the FTS logic constraint on features that identifies in which variants the transition has to be included;
- (2) `exit_state` is the state to which the transition points;
- (3) `dead` is a Boolean flag meant to mark the dead status of transitions;
- (4) `false_opt` is a Boolean flag meant to mark the false optional status of transitions.

It is worth to recall that, in accordance with Definition 2.2, we are interested only in paths starting in the initial state of the FTS; moreover, we focus on cycle-free paths that never cross twice the same state or the same transition.

*Definition 5.1 (path notations).* Let  $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$  be an FTS represented in above data structures and let  $p = s_0 t_1 s_1 \dots s_{n-1} t_n s_n$  ( $n \in \mathbb{N}$ ) be a cycle-free path of  $\mathcal{F}$ .

- Let  $p.\text{ENDSTATE}$  denote the final state of  $p$ , namely  $s_n$ .
- Let  $p.\text{ENDTRS}$  denote the final transition of  $p$ , namely  $t_n$ .
- Let  $\text{VsTED}(p)$  denote the predicate that holds whenever:  
 $s.\text{visited} = \text{true}$  iff  $s = s_i$ , for all  $i \leq n - 1$ .
- Let  $\text{SATED}(p)$  denote the predicate that holds exactly when  $\text{fts.fm} \wedge \text{bx}_p$  is satisfiable.
- Let  $\text{EXT}_p$  denote the set of all cycle-free paths extending  $p$  in a product, i.e., all paths  $p' = s_0 t_1 s_1 \dots s_{n-1} t_n s_n \dots t_{n+k} s_{n+k}$  ( $k \geq 0$ ) such that  $\text{SATED}(p')$  holds.

We remark that  $\text{VsTED}(p)$  predicates that the states marked as “visited” are exactly those in  $p$ , except for the last one. Moreover, as stated by the following proposition,  $\text{EXT}_p$  is closed under prefixes extending  $p$ .

*PROPOSITION 5.2 (EXT-CLOSURE).* Let  $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$  be an FTS and let  $p$  be one of its cycle-free path. If  $p' \in \text{EXT}_p$  then  $\text{EXT}_{p'} \subseteq \text{EXT}_p$ .

*PROOF.* If  $\text{SATED}(p)$  is false then  $\text{EXT}_p = \text{EXT}_{p'} = \emptyset$ . If there is  $p'' \in \text{EXT}_{p'}$  then  $p''$  extends  $p'$  and  $\text{SATED}(p'')$  holds. Clearly  $p''$  extends also  $p$ , so we conclude  $p'' \in \text{EXT}_p$ .  $\square$

We first prove correctness of the procedure `path_discover` from Listing 2.

*LEMMA 5.3 (CORRECTNESS OF PROCEDURE path\_discover).* Let  $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$  be the FTS represented in the algorithm data structures and let  $p = s_0 t_1 s_1 \dots s_{n-1} t_n s_n$  ( $n \in \mathbb{N}$ ) be a cycle-free path such that both  $\text{SATED}(p)$  and  $\text{VsTED}(p)$ .

- (1) The execution of `path_discover(p.endState, bxp)` recursively calls the function `path_discover(p'.endState, bxp')` in a situation in which  $\text{VsTED}(p')$  holds, for all and only  $p' \in \text{EXT}_p$  and  $p \neq p'$ .
- (2) The execution of `path_discover(fts.initial, true)` always ends in a finite number of steps.

- (3) Let  $p' \in \text{EXT}_p$  and  $s \in p'.\text{ENDSTATE}$ . If  $s.\text{exit\_trs} \neq \emptyset$  and  $\text{FM}_{\mathcal{F}} \Rightarrow (\text{bx}_p \Rightarrow (\forall t \in s.\text{EXIT\_TRS} t.\text{BX}))$  is not a tautology, then the execution of  $\text{path\_discover}(p.\text{endState}, \text{bx}_p)$  ensures that  $s.\text{live}$  is flagged false.
- (4) Let  $p' \in \text{EXT}_p$ ,  $s \in p'.\text{ENDSTATE}$ , and  $t \in s.\text{exit\_trs}$ . If  $\text{FM}_{\mathcal{F}} \Rightarrow (\text{bx}_{p'} \Rightarrow t.\text{BX})$  is not a tautology, then the execution of  $\text{path\_discover}(p.\text{endState}, \text{bx}_p)$  ensures that  $t.\text{false\_opt}$  is flagged false.
- (5) Let  $p' \in \text{EXT}_p$  be a path including at least a transition, and let  $t$  be  $p'.\text{ENDTRS}$ . The execution of  $\text{path\_discover}(p.\text{endState}, \text{bx}_p)$  ensures that  $t.\text{dead}$  is flagged false.

**PROOF.** (1) First we show that, if  $p' \in \text{EXT}_p$ , this implies a recursive call to  $\text{path\_discover}(p'.\text{endState}, \text{bx}_{p'})$  (in a situation in which  $\text{VsTED}(p')$  holds). The proof is by induction on the number of paths in  $\text{EXT}_p$ . If  $\text{EXT}_p = \emptyset$ , the proof is trivial; so, assume  $p' = s_0t_1s_1 \dots s_{n-1}t_ns_nt_{n+1}s_{n+1} \in \text{EXT}_p$  extending  $p$  with the unique additional transition  $t_{n+1}$ . Execution of  $\text{path\_discover}(p.\text{endState}, \text{bx}_p)$  is driven by the code of Listing 2. Line 2 predisposes a flag  $\text{visited}$  and makes  $\text{VsTED}(p')$  hold, thus a first statement requirement is satisfied. Then, since  $p'$  is cycle-free and  $\text{SATED}(p')$  holds, lines 11–12 concretely do the recursive call  $\text{path\_discover}(p'.\text{endState}, \text{bx}_{p'})$ . As  $\text{EXT}_{p'} \subseteq \text{EXT}_p$  by Proposition 5.2, the statement holds for all paths in  $\text{EXT}_{p'}$  by induction; so, the proof follows.  
Assume that, by executing  $\text{path\_discover}(p.\text{endState}, \text{bx}_p)$  a recursive call to  $\text{path\_discover}(p'.\text{endState}, \text{bx}_{p'})$  is done, in a situation in which  $\text{VsTED}(p')$  holds. The proof is done by induction on the number  $N$  of transitions of  $p'$  not in  $p$ . The call must be done by means of the lines 11–12 of Listing 2. If  $N = 1$  then line 11 ensures that  $p'$  is still a cycle-free path (properly extending  $p$ ) and  $\text{SATED}(p')$ ; namely  $p' \in \text{EXT}_p$ . If  $N \geq 1$  then the proof follows by induction.

- (2) As usual, we define FTSs without finiteness assumptions; but as for SPLs, their practical interest implicitly assumes that all constituents of  $\mathcal{F}$  are finite (this allows to store  $\mathcal{F}$  in our algorithm's data structure). First note that the number of transitions starting from the nodes of the FTS is bounded by  $\max_{s \in S} \#(s.\text{EXIT\_TRS}) \in \mathbb{N}$ . Thus, line 5 of the procedure in Listing 2 does a finite number of recursive calls. Next, note that the number of states  $\text{visited} == \text{false}$  is strictly decreased at each of the previous calls. Thus, the proof follows.
- (3) If  $p \neq p'$  then,  $\text{path\_discover}(p'.\text{endState}, \text{bx}_{p'})$  is recursively called by the point 1 of this lemma. Line 3 of the procedure prevents the update of the flag  $(p'.\text{endState}).\text{live}$  whenever it is already set to false. Line 4, if necessary, updates this flag in accordance to our statement.
- (4) If  $p \neq p'$  then,  $\text{path\_discover}(p'.\text{endState}, \text{bx}_{p'})$  is recursively called by the point 1 of this lemma. Line 6 of the procedure prevents the update of the flag  $(p'.\text{endtrs}).\text{false\_opt}$  whenever it is already set to false. Line 7, if necessary, updates this flag in accordance to our statement.
- (5)  $p' \in \text{EXT}_p$  ensures that  $\text{SATED}(p')$ , i.e.,  $p$  is a path such that  $\text{fts}.fm \wedge \text{bx}_p$  is satisfiable. Line 9 of the procedure stores true in  $\text{valid}_p$ . Line 10 prevents the update of the flag  $(p'.\text{endtrs}).\text{dead}$  whenever it is already false and, if necessary, updates this flag in accordance to our statement.  $\square$

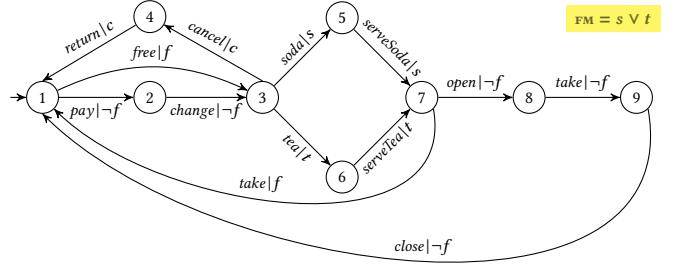


Figure 6: FTS of the vending machine from [27]

Next we prove correctness of the analysis algorithm in Listing 1.

**THEOREM 5.4 (CORRECTNESS OF ALGORITHM).**

Let  $\mathcal{F} = (S, \Sigma, s_0, \delta, F, \Lambda)$  be an FTS represented in the algorithm data structures. The execution of the ambiguities discovery algorithm (cf. Listing 1) sets the flags `dead`, `false_opt`, and `hDead` in the data structure in accordance with the fact that a transition is dead, a transition is false optional, and a state is not a hidden deadlock, respectively.

**PROOF.** The algorithm of Listing 1 begins by executing some initialisation, after which it calls  $\text{path\_discover}(\text{fts}.initial, \text{true})$ . Lemma 5.3 specifies the content of the flags `dead`, `false_opt`, and `live` following the aforementioned procedure call. The algorithm ends by setting the flags `s.hDead` as expected. Hence, the proof follows by Theorems 4.6, 4.9 and 4.12.  $\square$

## 6 ILLUSTRATIVE EXAMPLES

The python code allowing the verification of the examples presented in this section and in Sect. 8 is publicly available [7].

In Fig. 6, we depict an example FTS modelling the behaviour of a configurable vending machine selling soda and tea from [27], an FTS benchmark which was used in numerous publications, among which [5, 6, 11, 30, 32, 42–45, 47]. Its feature model can be represented by the formula  $s \vee t$  over the 4 features  $\{f, c, s, t\}$ , thus resulting in 12 products (i.e.,  $2^4 - 4$ , excluding  $\emptyset, \{f\}, \{c\}, \{f, c\}$ ). The FTS of the vending machine contains 9 states and 13 transitions.

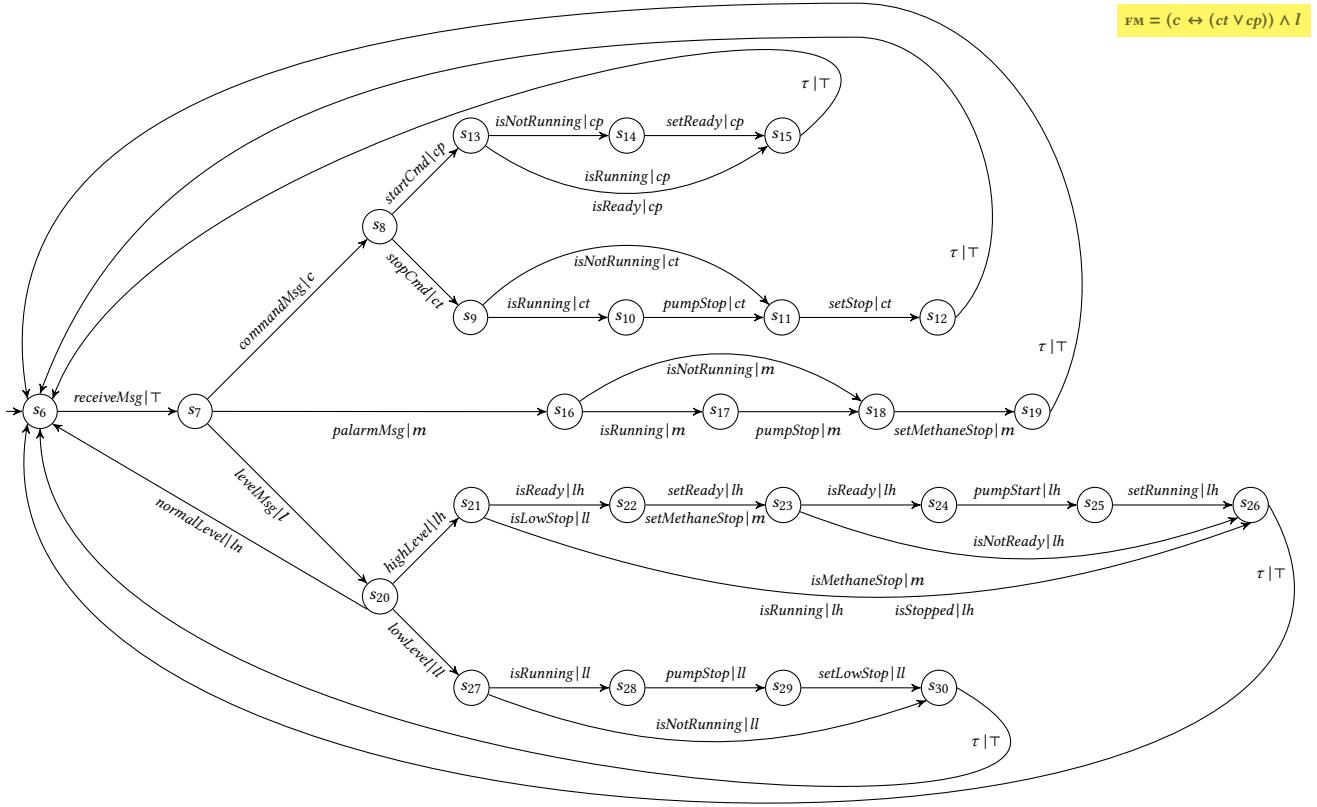
Listing 3 reports the result of applying static analysis to this FTS: no dead transitions and no hidden deadlocks, but 6 false optional transitions, viz.  $(2, \text{change}, \neg f, 3)$ ,  $(4, \text{return}, c, 1)$ ,  $(5, \text{serveSoda}, s, 7)$ ,  $(6, \text{serveTea}, t, 7)$ ,  $(8, \text{take}, \neg f, 9)$ , and  $(9, \text{close}, \neg f, 1)$ . Hence, the FTS is ambiguous, but it suffices to turn its false optional transitions into must transitions to make the FTS unambiguous.

### Listing 3: Result of the static analysis on the FTS of Fig. 6

```
Vending Machine: live
LIVE STATES = [1,2,3,4,5,6,7,8,9]
DEAD TRANSITIONS = []
FALSE OPTIONAL TRANSITIONS = [(2,3),(4,1),(5,7),(6,7),(8,9),(9,1)]
HIDDEN DEADLOCK STATES = []
```

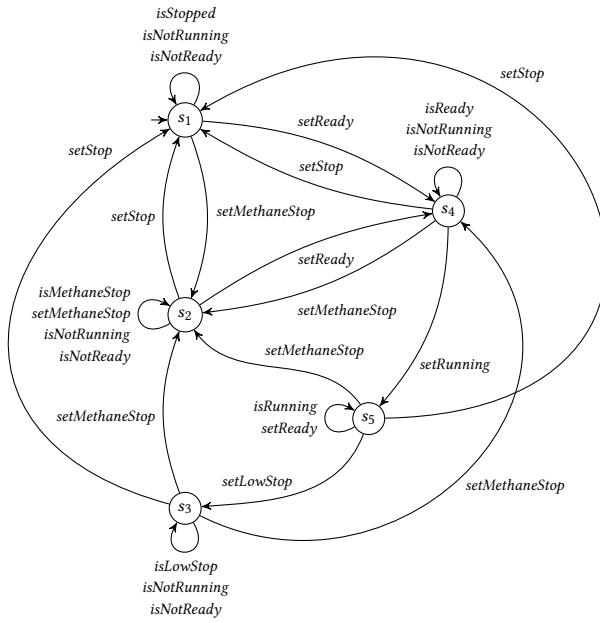
In Fig. 7, we depict an example FTS modelling the behaviour of the so-called *system* FTS that models the logic of a configurable controller of the mine pump model from [26, 27], a standard SPL benchmark for FTSs which was used in numerous publications, among which [10, 28, 30, 32, 35, 36, 43, 45–47]. The *system* FTS of this mine pump model contains 25 states and 41 transitions.<sup>1</sup>

<sup>1</sup>Transitions with more than one label are abbreviations for a transition for each label.



**Figure 7: The system FTS of the mine pump model from [27] (we have labelled transitions that were unlabelled with  $\tau | T$ )**

The controller of this mine pump model is the parallel composition of the system FTS with the so-called *state* FTS, depicted in Fig. 8.



**Figure 8: The state FTS of the mine pump model from [27]**

The mine pump has to keep a mine safe from flooding by pumping water from a shaft while avoiding a methane explosion. The reason we use the model from [26, 27] is that it comes with detailed FTSs. Its feature model can be represented by the formula  $\phi = (c \leftrightarrow (ct \vee cp)) \wedge l$  over the feature set  $F = \{c, ct, cp, m, l, ll, ln, lh\}$ , thus resulting in 64 products (i.e.,  $2^6$ , since  $\phi$  is equivalent to considering features  $\{ct, cp, m, ll, ln, lh\}$  to be optional).

Listing 4 reports the result of applying static analysis to the system FTS: no dead transitions, but numerous false optional transitions, among which  $(s_7, levelMsg, l, s_{20})$ , and one hidden deadlock state, viz.  $s_{20}$ . Indeed, state  $s_{20}$  is reachable in all products upon the execution of two must transitions (the second one being the false optional transition  $(s_7, levelMsg, l, s_{20})$ ), while  $s_{20}$  is a deadlock in all 8 products that lack any of the features from the subset  $\{ll, ln, lh\}$ .

#### Listing 4: Result of the static analysis on the FTS of Fig. 7

```

Mine Pump: not live
LIVE STATES = [S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16,S17,S18,S19,
S21,S22,S23,S24,S25,S26,S27,S28,S29,S30]
DEAD TRANSITIONS = []
FALSE OPTIONAL TRANSITIONS = [(S10,S11),(S11,S12),(S13,S14),
(S13,S15,isReady),(S13,S15,isRunning),(S14,S15),(S16,S17),
(S16,S18),(S17,S18),(S18,S19),(S21,S22,isReady),
(S21,S26,isRunning),(S21,S26,isStopped),(S22,S23,setReady),
(S23,S24),(S23,S26),(S24,S25),(S25,S26),(S27,S28),(S27,S30),
(S28,S29),(S29,S30),(S7,S20),(S9,S10),(S9,S11)]
HIDDEN DEADLOCK STATES = [S20]

```

Hence the *system* FTS is ambiguous, but it suffices to turn its false optional transitions into must transitions and to add an explicit deadlock state  $s_{\dagger}$  and a transition  $(s_{20}, \dagger, \neg ll \wedge \neg ln \wedge \neg lh, s_{\dagger})$  to make the *system* FTS unambiguous. Actually, a deadlock often indicates an error in the modelling, either in the feature model or in the behavioural model, i.e., the FTS. In fact, another solution to make the *system* FTS unambiguous is to slightly change the feature model, e.g., by requiring the presence of at least one of the features  $ll$ ,  $ln$ , or  $lh$  via an or-relationship. Doing so, the feature model becomes  $\phi = (c \leftrightarrow (ct \vee cp)) \wedge l \wedge (ll \vee ln \vee lh)$ , thus resulting in 56 products (i.e., excluding the 8 products over  $F$  that satisfy  $(c \leftrightarrow (ct \vee cp)) \wedge l$ , but lack any of the features from the subset  $\{ll, ln, lh\}$ ). In [26, 27, 32], instead, an alternative feature model in which only  $c$  (and implicitly  $ct$  and  $cp$ ) and  $m$  are optional is considered, resulting in only the four products over  $F$  that satisfy  $(c \leftrightarrow (ct \wedge cp)) \wedge l \wedge ll \wedge ln \wedge lh$ .

Yet another solution to make the *system* FTS unambiguous is to slightly change the FTS itself, to make sure that it contains neither a hidden nor an explicit deadlock state. In this case, it suffices to add one or more transitions to leave state  $s_{20}$  in a meaningful way. This is the solution opted for in [10, 30, 45–47], which use the specification in fPromela of the complete mine pump model (see below) as distributed with SNIP [28] (<http://projects.info.unamur.be/fts/snip/>) and ProVeLines [34] (<http://projects.info.unamur.be/fts/provelines/>) or translations thereof for mCRL2 [37] (<http://www.mcurl2.org/>) or VMC [17] (<http://fmlab.isti.cnr.it/vmc/>). Basically, three transitions are added to the *system* FTS of Fig. 7 from state  $s_{20}$  to the initial state  $s_6$  to cover the cases in which features from the subset  $\{ll, ln, lh\}$  are missing, viz.  $(s_{20}, highLevel, \neg lh, s_6)$ ,  $(s_{20}, lowLevel, \neg ll, s_6)$ , and  $(s_{20}, normalLevel, \neg ln, s_6)$ .<sup>2</sup>

In [26, 27], the controller of the mine pump model, composed of the *system* and *state* FTSs, interacts with an environment: it operates a water pump based on methane and water level sensors, modelled by three further FTSs. The parallel composition of these five FTSs is the above mentioned complete mine pump model. We depict the FTS of the methane level in Fig. 9 and refer to [26, 27] for the remaining FTSs. Moreover, *methaneRise* and *methaneLower* are local actions of this FTS that do not synchronise with any of the other four FTSs. Hence, while the solutions presented above make the *system* FTS of Fig. 7 unambiguous, it is immediately clear that the FTS of the complete mine pump model is deadlock-free, since it can indefinitely execute the sequence of actions *methaneRise* followed by *methaneLower*. This demonstrates the usefulness of analysing component FTSs in isolation. More on this in Sect. 8.

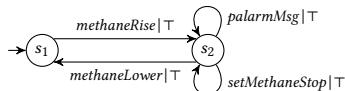


Figure 9: FTS of the methane level environment from [27]

## 7 USEFULNESS OF UNAMBIGUOUS FTSs

In analogy with anomaly detection in feature models, dead featured transitions in an FTS clearly indicate a modelling error, whereas false optional featured transitions often provide a wrong idea of the

<sup>2</sup>To satisfy the transition injectivity of Definition 2.3, in the FTS of Fig. 7 this results in the substitution of transition  $(s_{20}, normalLevel, ln, s_6)$  with  $(s_{20}, normalLevel, T, s_6)$ .

domain by giving the impression that certain behaviour is optional while actually it is mandatory (i.e., it occurs in all products of the FTS). However, the transformation of an ambiguous FTS into an unambiguous FTS also serves another purpose, viz. to facilitate family-based model checking of properties expressed in a fragment of the variability-aware action-based and state-based branching-time modal temporal logic v-ACTL and interpreted on so-called ‘live’ MTSs [5, 11–13]. A Modal Transition System (MTS) is an LTS that distinguishes admissible (‘may’), necessary (‘must’), and optional (may but not must) transitions such that by definition all necessary and optional transitions are also admissible [53, 54].

In [12], an MTS is defined to be *live* if all its states are live, where a live state of an MTS is such that it does not occur as a final state in any of its products (LTSs obtained from the MTS in a way similar to Definition 2.4), resulting in an MTS in which every path is infinite. Then it is proved that the validity of formulas expressed in a rich fragment of v-ACTL is preserved in all products (cf. Theorem 4 of [12]), thus allowing family-based model checking of MTSs.

It is not difficult to see that this result continues to hold for MTSs whose every state is either live or final.<sup>3</sup> Note that any FTS  $\mathcal{F}$  can be transformed into an MTS by considering its must transitions as necessary transitions, its featured transitions as optional transitions, and all its transitions as admissible, and by removing all feature expressions. If the FTS is unambiguous, then the corresponding MTS is live, with respect to the FTS’ set of products  $\text{Its}(\mathcal{F})$ , because it has no hidden deadlocks, and all transitions of the FTS that are mandatorily present in all products moreover correspond to must transitions in the MTS. This demonstrates that the above mentioned result from [12] can be carried over to unambiguous FTSs, thus allowing family-based model checking of such FTSs for the v-ACTL fragment v-ACTLive $^{\square}$ . Hence, the following result holds.

**PROPOSITION 7.1.** *Any formula  $\phi$  of v-ACTLive $^{\square}$  is preserved by unambiguous FTSs: given an unambiguous FTS  $\mathcal{F}$ , whenever  $\mathcal{F} \models \phi$ , then  $\mathcal{F}|_{\lambda} \models \phi$  for all products  $\mathcal{F}|_{\lambda} \in \text{Its}(\mathcal{F})$ .*

In the next section, we apply this result to an example FTS and provide examples of v-ACTLive $^{\square}$  formulas to illustrate its impact.

### 7.1 Family-Based Model Checking

We have seen in Sect. 3.2 how to transform an ambiguous FTS into an unambiguous one. Furthermore, as mentioned above, an FTS can be transformed into an MTS by considering its must transitions as necessary transitions, its featured transitions as optional transitions, and all its transitions as admissible, and by removing all feature expressions. In Fig. 10, we depict the MTS<sup>4</sup> that is obtained in this way from the unambiguous FTS (described in the beginning of Sect. 6) that corresponds to the FTS of Fig. 6.

As we argued in the beginning of this section, the resulting MTS is live, with respect to the FTS’ set of products, thus allowing family-based model checking for v-ACTLive $^{\square}$  (cf. Proposition 7.1). Example formulas of properties that can now be verified in a family-based manner on the MTS of Fig. 10 are the following:

- (1)  $AG AF_{pay \vee free} T$ : infinitely often, either action *pay* or action *free* is executed;

<sup>3</sup>Adding loops (labelled with a dummy symbol) to all final states makes the MTS live.

<sup>4</sup>Dashed edges depict optional transitions and solid edges depict necessary transitions.

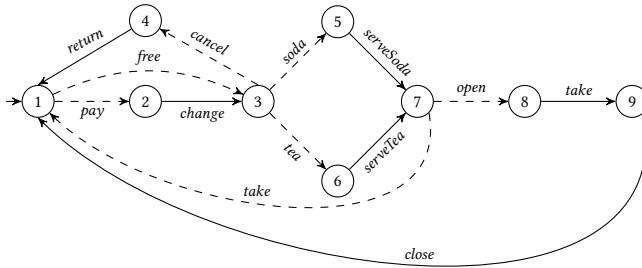


Figure 10: MTS obtained from the FTS of Fig. 6

- (2)  $AG [open] AF_{close} \top$ : it is always the case that the execution of action open is eventually followed by that of action close;
- (3)  $AG AF_{cancel \vee serveSoda \vee serveTea} \top$ : infinitely often, either action cancel or action serveSoda or action serveTea is executed;
- (4)  $\neg E [\top \neg tea U serveTea \top]$ : it is not possible that action serveTea is executed without being preceded by an execution of action tea;
- (5)  $[pay] AF_{take \vee cancel} \top$ : whenever action pay is executed, eventually also either action take or action cancel is executed.<sup>5</sup>

Such type of formulas can efficiently be verified with the variability model checker VMC, which is a tool for the analysis of behavioural SPL models specified as an MTS together with a set of logical variability constraints (akin to feature expressions) [17, 18]. VMC is the most recent member of the KandISTI product line of model checkers developed at ISTI-CNR over the past decades [13, 14]. The KandISTI toolset offers explicit-state on-the-fly model checking of functional properties expressed in specific action-based and state-based branching-time temporal logics derived from ACTL [41], which is the action-based version of the well-known logic CTL [25].

However, it is important to underline that VMC currently uses the variability constraints associated with the MTS to dynamically evaluate the liveness of each node. In this paper, we introduced a static analysis algorithm and transformation to turn any FTS into an unambiguous FTS, and we showed that this allows to establish a priori the liveness of all its nodes, and thus of the corresponding MTS. This makes it possible to improve VMC with the possibility to verify a live MTS without the need to use variability constraints to dynamically evaluate the liveness of its nodes. As a result, VMC could verify the above formulas in a family-based manner on the MTS of Fig. 10, since this MTS is live with respect to the set of products of the FTS of Fig. 6, meaning that whenever a formula holds for the MTS it also holds for all products defined by the FTS.

The static analysis algorithm presented in Sect. 5 and the FTS transformation defined in Sect. 3.2 could also give rise to a new KandISTI tool, tailored for family-based model checking of temporal logic properties on FTSs. At present, efficient SPL model checking against FTSs can be achieved by using dedicated family-based model checkers such as the ProVeLines [34] tool suite (including its predecessor SNIP [28]) or, alternatively, by using one of the highly optimised off-the-shelf classical model checkers such as SPIN or mCRL2, which have recently been made amenable to family-based SPL model checking against FTSs [10, 46].

<sup>5</sup>Abusing notation, this concerns execution of transition (8, take, 9), not of (7, take, 1).

## 8 SCALABILITY

From our experience, the bottleneck of the static analysis algorithm presented in this paper, as far as computational scalability is concerned, is the identification of all possible cycle-free paths of the FTS under scrutiny. While the static analyses performed on the example FTSs reported in Sect. 6 required a negligible amount of time, it is clear that this no longer holds for FTSs with hundreds of states and thousands of transitions. For instance, the FTS of the complete mine pump model of [26, 27], described in Sect. 6, has 417 states and 1255 transitions, which already are too many to efficiently visit all cycle-free paths in a depth-first manner. We plan to investigate optimisations of the static analysis algorithm that allow to reduce the impact of this bottleneck. We also plan to study the resulting algorithm's complexity.

However, it is not very likely that a system of a size like that of the complete mine pump is modelled as one monolithic FTS. Typically, a (large) system is designed in a modular way, as a composition of (smaller) components. A more promising strategy is thus to analyse FTS components in isolation and study the implications that ambiguities detected at the component level have for the entire system. This is confirmed by our analyses of the mine pump system in Sect. 6, where we have seen that the complete mine pump system is deadlock-free (i.e., live) even if the *system* FTS is not, simply because progress is guaranteed by the presence of a cycle of local actions in another component FTS.

In Table 1, we report some hard data concerning static analyses of the FTSs discussed so far. In addition, we report the results for two additional FTSs, viz. the configurable coffee machine from [3] depicted in Fig. 11, which is another FTS benchmark used in numerous publications, among which [8, 9, 12, 15, 16, 18, 19, 21, 26, 62], and the aforementioned controller of the mine pump model from [26, 27], i.e. the parallel composition of the *system* FTS with the *state* FTS.

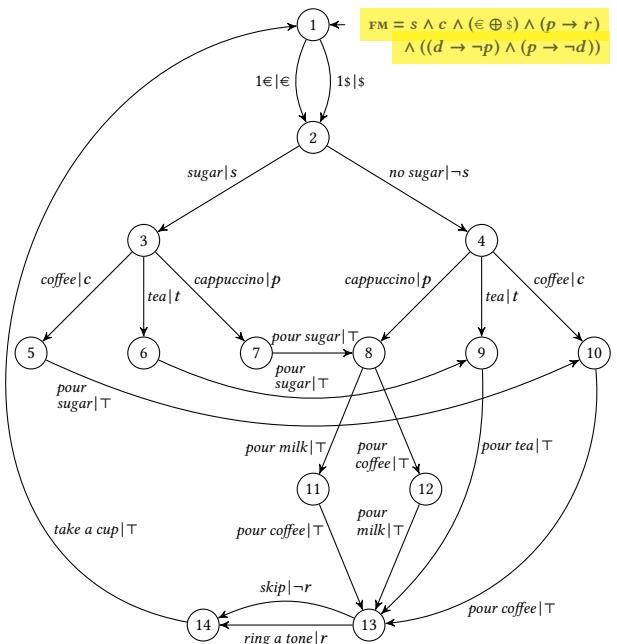


Figure 11: FTS of the coffee machine from [3]

**Table 1: Characteristics of the FTSs mentioned in this paper and results of applying static analysis**

FTS	characteristics				results of static analysis				computational effort	
	Model	# states	# transitions	# actions	live-ness	# dead transitions	# false optional transitions	# hidden deadlock states	run-time (s)	memory use (Mb)
Vending machine [27]		9	13	12	yes	0	6	0	0.68	41
Coffee machine [3]		14	22	14	yes	0	4	0	1.35	42
Mine pump (system) [27]		25	41	22	no	0	25	1	1.41	44
Mine pump (controller) [27]		77	104	22	no	0	59	4	5.37	48
Mine pump (complete) [27]		417	1255	26	yes	?	?	?	timeout	–

The experiments have been performed on a virtual machine<sup>6</sup> with 2048 Mb of allocated memory on a Windows 10 Pro 64 bit with 16 Gb of RAM and a CPU AMD Ryzen 7 1700X (8 core, 16 threads, 3.4 Ghz).

The FTSs of the vending machine (depicted in Fig. 6 and discussed in Sect. 6) and the coffee machine (depicted in Fig. 11) are both live (i.e., no deadlocks), with no dead transitions, while a respective 46% and 18% of their transitions are false optional. Their static analysis is immediate. Also that of the *system* FTS of the mine pump (depicted in Fig. 7 and discussed in Sect. 6) is immediate, but it is not live because one of its 25 states is a hidden deadlock. None of its transitions is dead, but 61% is false optional. Instead, it takes more than 5 s to analyse the FTS of the mine pump controller (i.e., the parallel composition of the *system* FTS and the *state* FTS (depicted in Fig. 8 and discussed in Sect. 6). It is not live, due to 4 hidden deadlock states and 57% of its transitions is false optional. Finally, from the discussion in Sect. 6 we know that the FTS of the complete mine pump, not depicted here, is live, but further results are missing, since we aborted the analysis after several hours.

We have not yet studied in detail what can be said about the preservation of ambiguities in a parallel composition of FTSs, either bottom-up, i.e., from the component FTSs to their parallel composition, or top-down, i.e., from the parallel composition to its constituting component FTSs. We conjecture that applying the static analysis algorithm to individual component FTSs, and consequently removing dead transitions and turning false optional transitions into must transitions, will not affect the behaviour of the parallel composition of these FTSs. On the contrary, it can be shown that in general the parallel composition of two unambiguous FTSs may result in a composed FTS with dead or false optional transitions, and, moreover, that the parallel composition may result in a composed FTS with more or less hidden deadlock states than in the individual component FTSs.

The application of the static analysis algorithm to individual component FTSs is surely desirable as it results in less ambiguous specifications of the components constituting a composed system, and it possibly also allows more efficient model checking of the composed system (cf. Sect. 7.1). Instead, the application of the static analysis algorithm to a composed FTS resulting from the parallel composition of several FTSs is less desirable for at least two reasons.

<sup>6</sup>Gentoo 201905, CLI Version VirtualBox (VDI) 64 bit, <https://www.osboxes.org/gentoo/>

On one hand, it risks to become problematic due to the exponential growth of the number of paths to be considered. On the other hand, the benefits of detecting ambiguities are greatly reduced because of the lack of a detailed specification of the composed FTS, which is merely a semantic model without a matching syntactic specification. Composed configurable systems can be described as Multi SPLs (MPLs), i.e., sets of interdependent SPLs [49]. It is not clear how to obtain results for composed FTSs by reusing results of analyses performed in isolation on its components, in analogy with recently proposed compositional approaches for analysing MPLs [38, 39, 55].

We conclude this section with two ideas to improve the algorithm, based on the fact that even though the number of paths in a graph can be exponentially larger than its size, several heuristic-based optimisations can be implemented to reduce the complexity of our algorithm in many cases. Firstly, paths must be computed only for transitions whose Boolean formula is not true (i.e., those that could be false optional or that cannot easily be stated as dead), and only for states whose live status is not trivial, i.e., states that have input and output transitions, and none of their output transitions' Boolean formula is true. For FTSs with very simple constraints on their transitions, this could greatly reduce the search space when traversing them. Secondly, checking the ambiguity criteria in a cycle-free FTS can be performed with a linear traversal, following the topological order of the FTS. It could be possible to extend this principle to FTSs with cycles, by using the topological order traversal only on the DAG subgraphs of an FTSs and thus greatly improving the efficiency of our algorithm on FTSs with few cycles.

## 9 CONCLUSION

In this paper, we have introduced several types of static analysis that can be performed over an FTS, we have given an effective algorithm for these analyses and demonstrated its correctness and completeness, and we have shown a number of example applications. The python code allowing to reproduce the verification of the examples presented Sect. 6 and Sect. 8 is publicly available [7].

In future work we plan to: (i) investigate the improvements suggested in the previous section, and (ii) evolve the prototype into a tool that supports to perform the static analysis of generic FTSs (specified in some standard format), to automatically disambiguate them, and to apply v-ACL model checking to generic FTSs.

## ACKNOWLEDGMENTS

We thank the four anonymous reviewers for useful comments and suggestions that helped us to improve the presentation.

## REFERENCES

- [1] Bowen Alpern and Fred B. Schneider. 1985. Defining liveness. *Inf. Process. Lett.* 4, 21 (1985), 181–185. [https://doi.org/10.1016/0020-0190\(85\)90056-0](https://doi.org/10.1016/0020-0190(85)90056-0)
- [2] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer. <https://doi.org/10.1007/978-3-642-37521-7>
- [3] Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, and Stefania Gnesi. 2011. Formal Description of Variability in Product Families. In *Proceedings of the 15th International Software Product Lines Conference (SPLC'11)*. IEEE, 130–139. <https://doi.org/10.1109/SPLC.2011.34>
- [4] Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary. 2016. An Adaptive Parallel SAT Solver. In *Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming (CP'16)* (LNCS), M. Rueher (Ed.), Vol. 9892. Springer, 30–48. [https://doi.org/10.1007/978-3-319-44953-1\\_3](https://doi.org/10.1007/978-3-319-44953-1_3)
- [5] Maurice H. ter Beek, Ferruccio Damiani, Stefania Gnesi, Franco Mazzanti, and Luca Paolini. 2015. From Featured Transition Systems to Modal Transition Systems with Variability Constraints. In *Proceedings of the 13th International Conference on Software Engineering and Formal Methods (SEFM'15)* (LNCS), R. Calinescu and B. Rümpe (Eds.), Vol. 9276. Springer, 344–359. [https://doi.org/10.1007/978-3-319-22969-0\\_24](https://doi.org/10.1007/978-3-319-22969-0_24)
- [6] Maurice H. ter Beek, Ferruccio Damiani, Stefania Gnesi, Franco Mazzanti, and Luca Paolini. 2019. On the Expressiveness of Modal Transition Systems with Variability Constraints. *Sci. Comput. Program.* 169 (2019), 1–17. <https://doi.org/10.1016/j.scico.2018.09.006>
- [7] Maurice H. ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini. 2019. Supplementary material for: “Static Analysis of Featured Transition Systems”. <https://doi.org/10.5281/zendodo.2616646>
- [8] Maurice H. ter Beek and Erik P. de Vink. 2014. Towards Modular Verification of Software Product Lines with mCRL2. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14)* (LNCS), Tiziana Margaria and Bernhard Steffen (Eds.), Vol. 8802. Springer, 368–385. [https://doi.org/10.1007/978-3-662-45234-9\\_26](https://doi.org/10.1007/978-3-662-45234-9_26)
- [9] Maurice H. ter Beek and Erik P. de Vink. 2014. Using mCRL2 for the Analysis of Software Product Lines. In *Proceedings of the 2nd FME Workshop on Formal Methods in Software Engineering (FormalISE'14)*. IEEE, 31–37. <https://doi.org/10.1145/2593489.2593493>
- [10] Maurice H. ter Beek, Erik P. de Vink, and Tim A. C. Willemse. 2017. Family-Based Model Checking with mCRL2. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE'17)* (LNCS), M. Huisman and J. Rubin (Eds.), Vol. 10202. Springer, 387–405. [https://doi.org/10.1007/978-3-662-54494-5\\_23](https://doi.org/10.1007/978-3-662-54494-5_23)
- [11] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. 2015. Using FMC for Family-based Analysis of Software Product Lines. In *Proceedings of the 19th International Software Product Line Conference (SPLC'15)*. ACM, 432–439. <https://doi.org/10.1145/2791060.2791118>
- [12] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. 2016. Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *J. Log. Algebr. Meth. Program.* 85, 2 (2016), 287–315. <https://doi.org/10.1016/j.jlamp.2015.11.006>
- [13] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. 2019. States and Events in KandISTI: A Retrospective. In *Models, Mindsets, Meta: The What, the How, and the Why Not?*, T. Margaria, S. Graf, and K. G. Larsen (Eds.). LNCS, Vol. 11200. Springer, 110–128. [https://doi.org/10.1007/978-3-030-22348-9\\_9](https://doi.org/10.1007/978-3-030-22348-9_9)
- [14] Maurice H. ter Beek, Stefania Gnesi, and Franco Mazzanti. 2015. From EU Projects to a Family of Model Checkers. In *Software, Services and Systems*, R. De Nicola and R. Hennicker (Eds.). LNCS, Vol. 8950. Springer, 312–328. [https://doi.org/10.1007/978-3-319-15545-6\\_20](https://doi.org/10.1007/978-3-319-15545-6_20)
- [15] Maurice H. ter Beek, Axel Legay, Alberto Lluch Lafuente, and Andrea Vandini. 2015. Quantitative Analysis of Probabilistic Models of Software Product Lines with Statistical Model Checking. *Electron. Proc. Theor. Comput. Sci.* 182 (2015), 56–70. <https://doi.org/10.4204/EPTCS.182.5>
- [16] Maurice H. ter Beek, Alberto Lluch Lafuente, and Marinella Petrocchi. 2013. Combining Declarative and Procedural Views in the Specification and Analysis of Product Families. In *Proceedings of the 17th International Software Product Line Conference (SPLC'13)*, Vol. 2. ACM, 10–17. <https://doi.org/10.1145/2499777.2500722>
- [17] Maurice H. ter Beek and Franco Mazzanti. 2014. VMC: Recent Advances and Challenges Ahead. In *Proceedings of the 18th International Software Product Line Conference (SPLC'14)*, Vol. 2. ACM, 70–77. <https://doi.org/10.1145/2647908.2655969>
- [18] Maurice H. ter Beek, Franco Mazzanti, and Aldi Sulova. 2012. VMC: A Tool for Product Variability Analysis. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12)* (LNCS), D. Giannakopoulou and D. Méry (Eds.), Vol. 7436. Springer, 450–454. [https://doi.org/10.1007/978-3-642-32759-9\\_36](https://doi.org/10.1007/978-3-642-32759-9_36)
- [19] Maurice H. ter Beek, Michel A. Reniers, and Erik P. de Vink. 2016. Supervisory Controller Synthesis for Product Lines Using CIF 3. In *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques (ISoLA'16)* (LNCS), T. Margaria and B. Steffen (Eds.), Vol. 9952. Springer, 856–873. [https://doi.org/10.1007/978-3-319-47166-2\\_59](https://doi.org/10.1007/978-3-319-47166-2_59)
- [20] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: a Literature Review. *Inf. Syst.* 35, 6 (2010), 615–636. <https://doi.org/10.1016/j.is.2010.01.001>
- [21] Harsh Beohar, Mahsa Varshosaz, and Mohammad Reza Mousavi. 2016. Basic behavioral models for software product lines: Expressiveness and testing pre-orders. *Sci. Comput. Program.* 123 (2016), 42–60. <https://doi.org/10.1016/j.scico.2015.06.005>
- [22] Nikolaj Björner, Anh-Dung Phan, and Lars Fleckenstein. 2015. vZ – An Optimizing SMT Solver. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)* (LNCS), C. Baier and C. Tinelli (Eds.), Vol. 9035. Springer, 194–199. [https://doi.org/10.1007/978-3-662-46681-0\\_14](https://doi.org/10.1007/978-3-662-46681-0_14)
- [23] Eric Bodden, Társis Tolédo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPL<sup>LIFT</sup> – Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proceedings of the 34th Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, 355–364. <https://doi.org/10.1145/2491956.2491976>
- [24] Brian Chess and Jacob West. 2007. *Secure Programming with Static Analysis*. Addison-Wesley.
- [25] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. 1986. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Sys.* 8, 2 (1986), 244–263. <https://doi.org/10.1145/5397.5399>
- [26] Andreas Classen. 2010. *Modelling with FTS: a Collection of Illustrative Examples*. Technical Report P-CS-TR SPLMC-00000001, University of Namur.
- [27] Andreas Classen. 2011. *Modelling and Model Checking Variability-Intensive Systems*. Ph.D. Dissertation, University of Namur.
- [28] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. 2012. Model checking software product lines with SNIP. *Int. J. Softw. Tools Technol. Transf.* 14, 5 (2012), 589–612. <https://doi.org/10.1007/s10009-012-0234-1>
- [29] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. 2014. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Sci. Comput. Program.* 80, B (2014), 416–439. <https://doi.org/10.1016/j.scico.2013.09.019>
- [30] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Trans. Softw. Eng.* 39, 8 (2013), 1069–1089. <https://doi.org/10.1109/TSE.2012.86>
- [31] Andreas Classen, Pierre Heymans, Pierre-Yves Schobbens, and Axel Legay. 2011. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, 321–330. <https://doi.org/10.1145/1985793.1985838>
- [32] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. 2010. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*. ACM, 335–344. <https://doi.org/10.1145/1806799.1806850>
- [33] Stephen A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual Symposium on Theory of Computing (STOC'71)*. ACM, 151–158. <https://doi.org/10.1145/800157.805047>
- [34] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. 2013. ProVeLines: A Product Line of Verifiers for Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference (SPLC'13)*, Vol. 2. ACM, 141–146. <https://doi.org/10.1145/2499777.2499781>
- [35] Maxime Cordy, Andreas Classen, Gilles Perrouin, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2012. Simulation-Based Abstractions for Software Product-Line Model Checking. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE, 672–682. <https://doi.org/10.1109/ICSE.2012.6227150>
- [36] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2013. Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-features. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. IEEE, 472–481. <https://doi.org/10.1109/ICSE.2013.6606593>
- [37] Sjoerd Cransen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. 2013. An Overview of the mCRL2 Toolset and Its Recent Advances. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*

- (TACAS'13) (LNCS), N. Piterman and S. A. Smolka (Eds.), Vol. 7795. Springer, 199–213. [https://doi.org/10.1007/978-3-642-36742-7\\_15](https://doi.org/10.1007/978-3-642-36742-7_15)
- [38] Ferruccio Damiani, Michael Lienhardt, and Luca Paolini. 2017. A Formal Model for Multi SPLs. In *Proceedings of the 7th International Conference on Fundamentals of Software Engineering (FSEN'17) (LNCS)*, M. Dastani and M. Sirjani (Eds.), Vol. 10522. Springer, 67–83. [https://doi.org/10.1007/978-3-319-68972-2\\_5](https://doi.org/10.1007/978-3-319-68972-2_5)
- [39] Ferruccio Damiani, Michael Lienhardt, and Luca Paolini. 2019. A formal model for Multi Software Product Lines. *Sci. Comput. Program.* 172 (2019), 203–231. <https://doi.org/10.1016/j.scico.2018.11.005>
- [40] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08) (LNCS)*, C. R. Ramakrishnan and J. Rehof (Eds.), Vol. 4963. Springer, 337–340. [https://doi.org/10.1007/978-3-540-78800-3\\_24](https://doi.org/10.1007/978-3-540-78800-3_24)
- [41] Rocco De Nicola and Frits W. Vaandrager. 1990. Action versus State based Logics for Transition Systems. In *Semantics of Systems of Concurrent Processes: Proceedings of the LITP Spring School on Theoretical Computer Science (LNCS)*, I. Guessarian (Ed.), Vol. 469. Springer, 407–419. [https://doi.org/10.1007/3-540-53479-2\\_17](https://doi.org/10.1007/3-540-53479-2_17)
- [42] Xavier Devroey, Gilles Perrouin, Axel Legay, Maxime Cordy, Pierre-Yves Schobbens, and Patrick Heymans. 2014. Coverage Criteria for Behavioural Testing of Software Product Lines. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (IsoLA'14) (LNCS)*, T. Margaria and B. Steffen (Eds.), Vol. 8802. Springer, 336–350. [https://doi.org/10.1007/978-3-662-45234-9\\_24](https://doi.org/10.1007/978-3-662-45234-9_24)
- [43] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2016. Featured Model-based Mutation Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. ACM, 655–666. <https://doi.org/10.1145/2884781.2884821>
- [44] Aleksandar S. Dimovski. 2018. Abstract Family-Based Model Checking Using Modal Featured Transition Systems: Preservation of CTL\*. In *Proceedings of the 21st International Conference on Fundamental Approaches to Software Engineering (FASE'18) (LNCS)*, A. Russo and A. Schürr (Eds.), Vol. 10802. Springer, 301–318. [https://doi.org/10.1007/978-3-319-89363-1\\_17](https://doi.org/10.1007/978-3-319-89363-1_17)
- [45] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wąsowski. 2017. Efficient family-based model checking via variability abstractions. *Int. J. Softw. Tools Technol. Transf.* 5, 19 (2017), 585–603. <https://doi.org/10.1007/s10009-016-0425-2>
- [46] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wąsowski. 2015. Family-Based Model Checking Without a Family-Based Model Checker. In *Proceedings of the 22nd International SPIN Symposium on Model Checking of Software (SPIN'15) (LNCS)*, B. Fischer and J. Geldenhuys (Eds.), Vol. 9232. Springer, 282–299. [https://doi.org/10.1007/978-3-319-23404-5\\_18](https://doi.org/10.1007/978-3-319-23404-5_18)
- [47] Aleksandar S. Dimovski and Andrzej Wąsowski. 2017. Variability-Specific Abstraction Refinement for Family-Based Model Checking. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE'17) (LNCS)*, M. Huisman and J. Rubin (Eds.), Vol. 10202. Springer, 406–423. [https://doi.org/10.1007/978-3-662-54494-5\\_24](https://doi.org/10.1007/978-3-662-54494-5_24)
- [48] Marijn Heule, Matti Järvisalo, and Martin Suda. [n.d.]. The international SAT Competitions web page. <https://www.satcompetition.org/>. Accessed: 2019-03-22.
- [49] Gerald Holl, Paul Grünbacher, and Rick Rabiser. 2012. A systematic review and an expert survey on capabilities supporting multi product lines. *Inf. Softw. Technol.* 54, 8 (2012), 828–852. <https://doi.org/10.1016/j.infsof.2012.02.002>
- [50] Frank Hutter, Marius Lindauer, Adrian Balint, Sam Bayless, Holger Hoos, and Kevin Leyton-Brown. 2017. The Configurable SAT Solver Challenge (CSSC). *Artifi. Intell.* 243 (2017), 1–25. <https://doi.org/10.1016/j.artint.2016.09.006>
- [51] Christian Kästner and Sven Apel. 2008. Type-checking Software Product Lines – A Formal Approach. In *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE'08)*. IEEE, 258–267. <https://doi.org/10.1109/ASE.2008.36>
- [52] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. 2011. Reducing Combinatorics in Testing Product Lines. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD'11)*. ACM, 57–68. <https://doi.org/10.1145/1960275.1960284>
- [53] Jan Křetinský. 2017. 30 Years of Modal Transition Systems: Survey of Extensions and Analysis. In *Models, Algorithms, Logics and Tools*, L. Aceto, G. Bacci, G. Bacci, A. Ingólfssdóttir, A. Legay, and R. Mardare (Eds.). LNCS, Vol. 10460. Springer, 36–74. [https://doi.org/10.1007/978-3-319-63121-9\\_3](https://doi.org/10.1007/978-3-319-63121-9_3)
- [54] Kim Guldstrand Larsen and Bent Thomsen. 1988. A Modal Process Logic. In *Proceedings of the 3rd Symposium on Logic in Computer Science (LICS'88)*. IEEE, 203–210. <https://doi.org/10.1109/LICS.1988.5119>
- [55] Michael Lienhardt, Ferruccio Damiani, Simone Donetti, and Luca Paolini. 2018. Multi Software Product Lines in the Wild. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'18)*. ACM, 89–96. <https://doi.org/10.1145/3168365.3170425>
- [56] Zohar Manni and Amir Pnueli. 1995. *Temporal Verification of Reactive Systems: Safety*. Springer. <https://doi.org/10.1007/978-1-4612-4222-2>
- [57] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*. ACM, 231–240.
- [58] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 2005. *Principles of Program Analysis*. Springer. <https://doi.org/10.1007/978-3-662-03811-6>
- [59] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 27, 4 (2018), 18:1–18:33. <https://doi.org/10.1145/3280986>
- [60] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (2014), 6:1–6:45. <https://doi.org/10.1145/2580950>
- [61] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Sci. Comput. Program.* 79 (2014), 70–85. <https://doi.org/10.1016/j.scico.2012.06.002>
- [62] Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi. 2018. Basic behavioral models for software product lines: Revisited. *Sci. Comput. Program.* 168 (2018), 171–185. <https://doi.org/10.1016/j.scico.2018.09.001>

# Learning from Difference: An Automated Approach for Learning Family Models from Software Product Lines

Carlos Diego N. Damasceno  
damascenodiego@usp.br  
University of São Paulo, BR and  
University of Leicester, UK

Mohammad Reza Mousavi  
mm789@leicester.ac.uk  
University of Leicester  
Leicester, UK

Adenilso Simao  
adenilso@icmc.usp.br  
University of São Paulo (ICMC-USP)  
São Carlos, SP, BR

## ABSTRACT

Substantial effort has been spent on extending specification notations and their associated reasoning techniques to software product lines (SPLs). Family-based analysis techniques operate on a single artifact, referred to as a family model, that is annotated with variability constraints. This modeling approach paves the way for efficient model-based testing and model checking for SPLs. Albeit reasonably efficient, the creation and maintenance of family models tend to be time consuming and error-prone, especially if there are crosscutting features. To tackle this issue, we introduce  $FFSM_{Diff}$ , a fully automated technique to learn featured finite state machines (FFSM), a family-based formalism that unifies Mealy Machines from SPLs into a single representation. Our technique incorporates variability to compare and merge Mealy machines and annotate states and transitions with feature constraints. We evaluate our technique using 34 products derived from three different SPLs. Our results support the hypothesis that families of Mealy machines can be effectively merged into succinct FFSMs with fewer states, especially if there is high feature sharing among products. These indicate that  $FFSM_{Diff}$  is an efficient family-based model learning technique.

## CCS CONCEPTS

- Networks → Formal specifications; • Theory of computation → Query learning; • Hardware → Finite state machines;
- Software and its engineering → Software product lines.

## KEYWORDS

Software product lines, Model learning, Family model, 150% model

### ACM Reference Format:

Carlos Diego N. Damasceno, Mohammad Reza Mousavi, and Adenilso Simao. 2019. Learning from Difference: An Automated Approach for Learning Family Models from Software Product Lines. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3336294.3336307>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.  
ACM ISBN 978-1-4503-7138-4/19/09...\$15.00  
<https://doi.org/10.1145/3336294.3336307>

## 1 INTRODUCTION

Analysis and maintenance of software product lines (SPL) are known to be challenging [58]; they should avoid repetitive analysis of shared assets, and at the same time cater for possible feature interactions [6]. To tackle these issues, substantial effort has been spent on extending specification notations and their associated reasoning techniques to SPLs [13, 20, 28, 29] leading to efficient family-based analysis [58].

Family-based analysis operates on a single specification, referred to as *family model*, that is annotated with feature constraints as propositional logic formulae to express the combination of features involved in the concerned part of the model [58]. Thus, using SAT solvers [42], family models are amenable to family model-based testing [61] and family model checking [10] where redundant analysis of shared assets are avoided or minimised. Moreover, the cost of family-based analysis is mainly determined by the number and size of features and the amount of feature sharing, rather than the number of valid products [58].

Model-based techniques specifically tailored to family models have enabled test generation [9, 14, 29] and model checking [52, 57] at reduced cost. Nevertheless, the creation and maintenance of test models are known to be difficult, time consuming and error-prone [61], and the traceability between the family- and variability models can be complex due to crosscutting features [54]. Added to this, as requirements change and product instances evolve, the lack of maintenance may render family models outdated [68].

Motivated by these issues, in this paper we discuss how the creation and maintenance of family models can be performed by combining techniques for automata learning [62], feature model analysis [11], and automated comparison of state-based models [69]. Thus, we introduce  $FFSM_{Diff}$ , a fully-automated technique to learn family models by comparing and merging product models.

Our technique extends an approach for comparing labeled transition systems (LTS) [69] to family models, i.e., featured finite state machines (FFSM) [26, 29], by incorporating variability to express product-specific behaviors with feature constraints. An FFSM is a family-based formalism that combines the Mealy Machines [16] of all products of an SPLs into a single model by annotating states and transitions with feature constraints [26]. To evaluate our approach, we designed the following research questions (RQ):

- (RQ1) Is our automated technique effective in learning succinct family models compared to the total size of the products?
- (RQ2) Is the size of learnt family models influenced by the amount of feature reuse?
- (RQ3) Is our automated technique effective in learning succinct family models compared to hand-crafted family models?

In our evaluation, we used 34 Mealy machines derived from three SPLs of previous studies [19, 29]. Although these case studies are abstract representations of SPLs, they comprise many non-trivial aspects, such as the possibility of infinite behaviour and the existence of states with similar or identical behaviour in different products [69].

As a measure of succinctness, we use the average size of learnt FFSMs compared to the average total size of the products under learning. We describe *size* in terms of the *number of states* as it is one of the factors that influences the complexity of model-based testing [16], model checking [10], and model learning [62]. Thus, learning succinct models can lead to efficient analysis.

We used the Mann-Whitney test to check if there was significant difference ( $p < 0.01$ ) between the sizes of the FFSMs and products and used the Vargha-Delaney's  $\hat{A}$  effect size [64] to assess the likelihood of the learnt FFSM being more succinct [7]. To evaluate if the amount of feature reuse influenced the size of the learnt FFSMs, we used Pearson's correlation coefficient.

Our results indicate that families of Mealy machines can be effectively combined into a succinct FFSM, i.e., with far fewer states than the total number of states in all products under learning. Moreover, we also show that there is a strong negative correlation between the amount of feature reuse and the size of learnt FFSMs. Thus, FFSMs learnt from similar products tend to have fewer states than those built from drastically different products. These results show that our technique is an efficient family-based technique for automatically learning behavioral models of SPLs. Our approach can be helpful to domain engineering by supporting the inclusion of new application requirements, SPL re-engineering [22], evolution [43], and traceability analysis [63].

Thus, our contributions are threefold: (1) we introduce a technique to automate the process of building family models by means of comparing FSMs and analyzing feature models; (2) we present an experiment evaluating our technique and showing its effectiveness for learning FFSMs; and (3) we show that amount of feature reuse is a factor that affects family model learning.

The rest of this paper is organized as follows: In section 2, we briefly discuss software product lines (SPL), featured finite state machines (FFSM) and an approach to compare state-based models [69]. In section 3, we introduce our *FFSM<sub>Diff</sub>* algorithm to learn FFSMs from products specifications. In section 4, we present our experiment design and artifacts, lab package structure<sup>1</sup>, the analysis of results and threats to validity. In section 5, we discuss related work. In section 6, we close this paper with our conclusions and the directions of our future work.

## 2 PRELIMINARIES

### 2.1 Software Product Lines

A software product line (SPL) is a family of products sharing a common and managed set of *features* that are developed in a prescribed way to satisfy the specific needs of a particular market segment. Let  $F$  be the set of features of an SPL. A product  $p$  is defined by a set of features  $p \subseteq F$  from a feature model  $FM$  [39].

<sup>1</sup> The lab package is available at <https://github.com/damascenodiego/learningFFSM>

A feature model  $FM$  captures all information about common and variant features of an SPL as a hierarchically arranged set of interconnected features. Based on a feature model, the powerset  $\mathcal{P}(F)$  of all feature combinations is constrained to a subset of valid products  $P \subseteq \mathcal{P}(F)$  that satisfy its feature constraints.

Feature constraints are propositional logic formulae that interpret the elements from  $F$  in terms of propositional variables. SAT solvers [42] can be used to detect valid feature models or feature combinations, core features (i.e., features that are part of all products) and redundancies in feature model [11]. We denote by  $B(F)$  the set of all feature constraints. The subset  $\Lambda \subseteq B(F)$  defines all valid product configurations of an SPL.

A product configuration  $\rho \in B(F)$  of a product  $p \in P$  is a feature constraint that expresses the conjunction of all features in  $p$  and the conjunction of negated features that are absent from it, i.e.,  $\rho = (\bigwedge_{f \in p} f) \wedge (\bigwedge_{f \notin p} \neg f)$ . Given a feature constraint  $\chi \in B(F)$ , a product configuration  $\rho \in \Lambda$  satisfies  $\chi$ , denoted by  $\chi \models \rho$ , iff the feature constraint  $\chi \wedge \rho$  is *true*. To illustrate these concepts, we use the Arcade Game Maker SPL.

**Example 2.1.** (The Arcade Game Maker SPL) The Arcade Game Maker (AGM) SPL has three alternative features (i.e., Brickle, Pong and Bowling) and one optional feature (i.e., Save). The feature model depicted in Figure 1 has six valid product configurations, among which three satisfy the feature constraint  $\neg S$ .

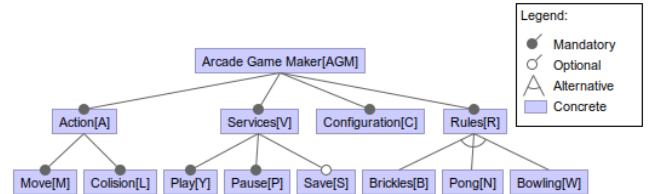


Figure 1: The AGM feature model

Given two feature constraints  $\omega_a$  and  $\omega_b$  from a feature model  $FM$ , and  $\Lambda_a, \Lambda_b \subseteq \Lambda$  satisfying  $\omega_a$  and  $\omega_b$ , respectively, we say that  $\omega_a, \omega_b$  are equivalent under  $FM$  if  $\Lambda_a = \Lambda_b$ .

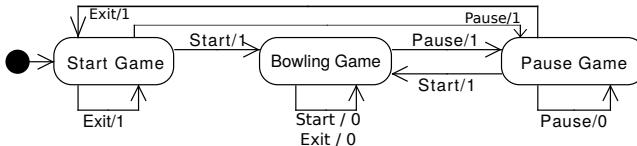
### 2.2 Featured Finite State Machines

Featured Finite State Machines are extensions of Finite State Machines [27], defined below, with feature constraints.

**DEFINITION 2.1.** (*Finite state machine*) A *finite state machine* (FSM) is a septuple  $M = \langle S, s_0, I, O, D, \delta, \lambda \rangle$  where  $S$  is the finite set of states,  $s_0 \in S$  is the initial state,  $I$  and  $O$  are the sets of inputs and outputs, respectively,  $D \subseteq S \times I$  is the specification domain, and  $\delta : D \rightarrow S$  and  $\lambda : D \rightarrow O$  are the transition and output functions.

Initially, an FSM is in the initial state  $s_0$ . Given a current state  $s_i \in S$ , when a defined input  $x \in I$ , such that  $(s_i, x) \in D$ , is applied, the FSM responds by moving to state  $s_j = \delta(s_i, x)$  and producing output  $y = \lambda(s_i, x)$ . The concatenation of two input sequences  $\alpha$  and  $\omega$  is denoted by  $\alpha \cdot \omega$ . An input sequence  $\alpha = x_1 \cdot x_2 \cdot \dots \cdot x_n \in I^*$  is defined in state  $s \in S$  if there are states  $s_1, s_2, \dots, s_{n+1}$  such that  $s = s_1$  and  $\delta(s_i, x_i) = s_{i+1}$ , for all  $1 \leq i \leq n$ . Transition are often represented as tuples  $(s_i, x, y, s_j)$  with the origin state, input, output, and destination states, respectively; or by directed edges labeled with input and output symbols, i.e.,  $s_i \xrightarrow{i/o} s_j$ .

**Example 2.2.** (Example of Mealy machine) In Figure 2, we show an example of an FSM describing a product from the AGM SPL. In this example, we have  $S = \{Start\ Game, Bowling\ Game, Pause\ Game\}$ ,  $I = \{Start, Pause, Exit\}$  and  $O = \{0, 1\}$ .



**Figure 2: Example of FSM [29]**

Transition and output functions are lifted to sequences of input in the standard way. Namely, for the empty input sequence  $\epsilon$ ,  $\delta(s, \epsilon) = s$  and  $\lambda(s, \epsilon) = \epsilon$ . For an input  $\alpha \cdot x$  defined in state  $s$ , we have  $\delta(s, \alpha \cdot x) = \delta(\delta(s, \alpha), x)$  and  $\lambda(s, \alpha \cdot x) = \lambda(s, \alpha)\lambda(\delta(s, \alpha), x)$ . An input sequence  $\alpha$  is a prefix of  $\beta$ , denoted by  $\alpha \leq \beta$ , if  $\beta = \alpha \cdot \omega$ , for some sequence  $\omega$ . An input sequence  $\alpha$  is a proper prefix of  $\beta$ , denoted by  $\alpha < \beta$ , if  $\beta = \alpha \cdot \omega$ , for  $\omega \neq \epsilon$ . The prefixes of a set  $T$  of input sequences are denoted by  $\text{pref}(T) = \{\alpha \mid \exists \beta \in T, \alpha < \beta\}$ . If  $T = \text{pref}(T)$ , it is *prefix-closed*.

An input sequence  $\alpha \in I^*$  is a transfer sequence from  $s$  to  $s'$  if  $\delta(s, \alpha) = s'$ . An input sequence  $\gamma$  is a separating sequence for  $s_i, s_j \in S$  if  $\lambda(s_i, \gamma) \neq \lambda(s_j, \gamma)$ . Two states  $s_i, s_j \in S$  are equivalent if for all  $\alpha \in I$ ,  $\lambda(s_i, \alpha) = \lambda(s_j, \alpha)$ , otherwise they are distinguishable. An FSM is *complete* if  $D = S \times I$ , otherwise it is *partial*.

An FSM is *deterministic* if, for each state  $s_i$  and input  $x$ , there is at most one possible state  $s_j = \delta(s_i, x)$  and output  $y = \lambda(s_i, x)$ . If all states of an FSM are pairwise distinguishable, it is *minimal*. If all states of an FSM are reachable from  $s_0$ , it is *initially connected*. If every state is reachable from all states, it is *strongly connected*. In this study, we focus on complete, deterministic, minimal, and initially connected FSMs, which are hereafter called finite state machines.

**DEFINITION 2.2 (FEATURED FINITE STATE MACHINE).** An FFSM is a septuple  $\langle F, \Lambda, C, c_0, Y, O, \Gamma \rangle$ , where: (i)  $F$  is a finite set of features, (ii)  $\Lambda$  is the set of product configurations, (iii)  $C \subseteq S \times B(F)$  is a finite set of conditional states, where  $S$  is a finite set of state labels,  $B(F)$  is the set of all feature constraints, and  $C$  satisfies the condition:

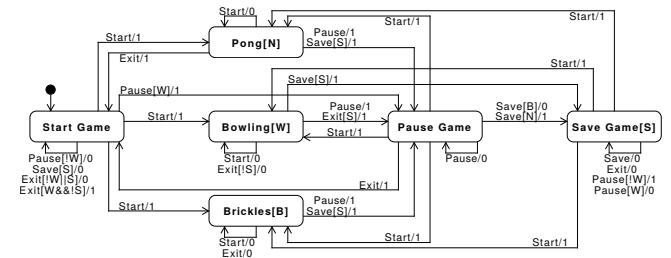
$$\forall (s, \phi) \in C, \exists \rho \in \Lambda | \rho \models \phi \quad (1)$$

(iv)  $c_0 = (s_0, \text{true}) \in C$  is the initial conditional state of the FFSM, (v)  $Y \subseteq I \times B(F)$  is a finite set of conditional inputs, where  $I$  is the finite set of input symbols, (vi)  $O$  is the finite set of output symbols, and (vii)  $\Gamma \subseteq C \times Y \times O \times C$  is the set of conditional transitions satisfying the condition:

$$\forall((s, \phi), (x, \phi''), o, (s', \phi')) \in \Gamma, \exists \rho \in \Lambda | \rho \models (\phi \wedge \phi' \wedge \phi'') \quad (2)$$

The conditions (1) and (2) ensure that all conditional states and transitions are present in at least one valid product of the SPL. A conditional state  $c = (s, \phi) \in C$  is alternatively denoted by  $s[\phi]$ .

A conditional transition  $(c, (x, \phi), o, c')$  from conditional state  $c$  to  $c'$  with conditional input  $x$  and output  $o$  is alternatively denoted  $x[\phi]/o$ . The logical operators and, or and not are denoted by the symbols  $\&$ ,  $|$ , and  $\neg$ , respectively. An omitted condition means that the condition is *true*.



Structurally comparing two state machines is a difficult task which involves establishing equivalence relationships between states and transitions. To achieve this goal, Walkinshaw and Bogdanov [69] proposed  $LTS_{Diff}$ , an algorithm to compute the precise difference between two state machines. In this section, we discuss the  $LTS_{Diff}$  algorithm in terms of FSMs.

$$\begin{aligned} \text{Succ}_{a,b} = & \{(c,d,i,o) \in S_r \times S_u \times (I_r \cup I_u) \times (O_r \cup O_u), \text{ such that} \\ & \delta_r(a,i) = c, \delta_u(b,i) = d, \text{ and} \\ & \lambda_r(a,i) = \lambda_u(b,i) = o\} \end{aligned}$$

Second, a global similarity score is calculated by aggregating the scores of states connected to the original pair as follows:

$$S_{\text{Succ}}^G(a, b) = \frac{1}{2} \frac{\sum_{(c, d, i, o) \in \text{Succ}_{a,b}} (1 + k \times S_{\text{Succ}}^G(c, d))}{|\sum_r^{\text{out}}(a) - \sum_u^{\text{out}}(b)| + |\sum_r^{\text{out}}(b) - \sum_u^{\text{out}}(a)| + |\text{Succ}_{a,b}|}$$

An attenuation ratio  $k$  is used to give precedence to state pairs that are closer to the original pair of states and the notation  $\sum_r^{\text{out}}(a)$  refers to the set of labels of outgoing transitions for state  $a$  of  $M_r$ . Thus, the expression  $|\sum_r^{\text{out}}(a) - \sum_u^{\text{out}}(b)| + |\sum_r^{\text{out}}(b) - \sum_u^{\text{out}}(a)|$  denotes the number of outgoing transitions from both states  $a$  and  $b$  that do not match each other.

Given two FSMs  $M_r$  and  $M_u$ , the global similarity score  $S_{\text{Succ}}^G(a, b)$  is used to build a system of linear equations, such that each equation corresponds to the  $S_{\text{Succ}}^G(a, b)$  for one specific pair of states  $(a, b) \in S_r \times S_u$ .

**Example 2.4.** (Illustration of a system of linear equations) In Table 1, we depict the system of equations resulting from the comparison of the alternative products from the AGM SPL in Figures 2 and 4. State pairs are represented by the first two letters of their respective names.

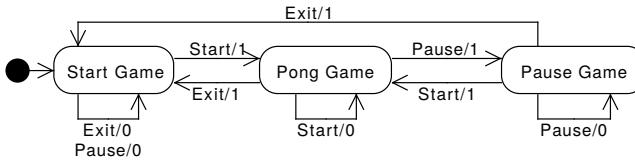


Figure 4: FSM of an alternative product from the AGM SPL

Table 1: Illustration of a system of linear equations

Pair	(St,St)	(St,Po)	(St,Pa)	(Bo,St)	(Bo,Po)	(Bo,Pa)	(Pa,St)	(Pa,Po)	(Pa,Pa)	
(St,St)	5.0	0.0	0.0	0.0	-0.5	0.0	0.0	0.0	0.0	3
(St,Po)	0.0	12.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
(St,Pa)	0.0	0.0	7.5	0.0	-0.5	0.0	0.0	0.0	0.0	2
(Bo,St)	0.0	0.0	0.0	9.5	0.0	0.0	0.0	0.0	0.0	1
(Bo,Po)	0.0	0.0	0.0	0.0	7.5	0.0	0.0	0.0	-0.5	2
(Bo,Pa)	0.0	0.0	0.0	0.0	0.0	12.0	0.0	0.0	0.0	0
(Pa,St)	0.0	0.0	0.0	0.0	-0.5	0.0	7.5	0.0	0.0	2
(Pa,Po)	-0.5	0.0	0.0	0.0	0.0	0.0	0.0	10.0	0.0	1
(Pa,Pa)	-0.5	0.0	0.0	0.0	-0.5	0.0	0.0	0.0	5.5	3

The global similarity is calculated both in terms of future behavior (i.e., outgoing transitions) and past behaviors (i.e., incoming transitions). The global similarity score for incoming transitions  $S_{\text{Prev}}^G(a, b)$  is calculated in a similar manner.

Consider the systems of equations for  $S_{\text{Succ}}^G(a, b)$  and  $S_{\text{Prev}}^G(a, b)$ , the similarity scores for each pair  $(a, b)$  are averaged as follows:

$$S(a, b) = \frac{S_{\text{Succ}}^G(a, b) + S_{\text{Prev}}^G(a, b)}{2}$$

### Algorithm 1: The $LTS_{\text{Diff}}$ algorithm

```

1 Input: FSM  $M_r$ , FSM  $M_u$ ,  $k$ ,  $t$ ,  $r$ ;
2  $PairsToScore \leftarrow computeScores(M_r, M_u, k);$ 
3  $KPairs \leftarrow identifyLandmarks(PairsToScore, t, r);$ 
4 if  $KPairs = \emptyset$  and  $S(s_{0_r}, s_{0_u}) > 0$  then
5    $| KPairs \leftarrow (s_{0_r}, s_{0_u});$ 
6 end
7  $NPairs \leftarrow \cup_{(a,b) \in KPairs} Surr(a, b) - KPairs;$ 
8 while  $NPairs \neq \emptyset$  do
9   while  $NPairs \neq \emptyset$  do
10    |  $(a, b) \leftarrow pickHighest(NPairs, PairsToScore);$ 
11    |  $KPairs \leftarrow KPairs \cup (a, b);$ 
12    |  $NPairs \leftarrow removeConflicts(NPairs, (a, b));$ 
13  end
14   $NPairs \leftarrow \cup_{(a,b) \in KPairs} Surr(a, b) - KPairs;$ 
15 end
16 return( $KPairs$ );

```

2.3.2 *The  $LTS_{\text{Diff}}$  algorithm.* Given the averaged scores, the comparison of two models follows a similar process to how humans navigate through an unfamiliar landscape with a map [69]. This process is shown in Algorithm 1.

First, a filtering method denoted by *identifyLandmarks* selects the top  $t\%$  most equivalent pairs, and, if one state is matched to several others, a ratio  $r$  includes only those pairs that are at least  $r$  times as good as any other match. If no state is selected, then the initial states are selected as initial landmarks. Second, the algorithm proceeds from the initial landmarks using the *Surr*( $a, b$ ) function to reach the surrounding states through matching incoming and outgoing transitions and adds them to a set of candidate matched state pairs  $NPairs$ . Third, the set  $NPairs$  is iterated in the order of similarity scores. Once a pair  $(a, b)$  is selected, it is added to a set of confirmed matches  $KPairs$  and all elements in  $NPairs$  that include either  $a$  or  $b$  are discarded. This process iterates until  $NPairs$  becomes empty.

## 3 THE FFSM<sub>Diff</sub> ALGORITHM

In this section, we introduce the *FFSM<sub>Diff</sub>* algorithm, a fully automated technique that combines FSMs into one succinct FFSM [26, 29] and annotates states and transitions with feature constraints. Although our technique is discussed in terms of FFSMs, it can be extended to other family-based notations [12], such as featured transition systems (FTS) [13, 20].

Essentially, *FFSM<sub>Diff</sub>* allows to learn an FFSM model (i) from two FSMs, or (ii) including an FSM into an existing FFSM. The former approach is applicable when there is no FFSM existing a priori and the latter if there is a new configuration  $\rho_u \notin \Lambda_r$  not included in an FFSM  $FF_r$  specifying a set of configurations  $\Lambda_r$ , respectively. In both cases, we assume that the feature model of the SPL and the configurations of the products under learning are known.

### 3.1 Learning a fresh FFSM

Consider two FSM models  $M_r = \langle S_r, s_{0_r}, I_r, O_r, D_r, \delta_r, \lambda_r \rangle$  and  $M_u = \langle S_u, s_{0_u}, I_u, O_u, D_u, \delta_u, \lambda_u \rangle$  specifying products  $p_r$  and  $p_u$  that implement configurations  $\rho_r = (\bigwedge_{f \in p_r} f) \wedge (\bigwedge_{f \notin p_r} \neg f)$  and  $\rho_u = (\bigwedge_{f \in p_u} f) \wedge (\bigwedge_{f \notin p_u} \neg f)$ . To learn a fresh FFSM from  $M_r$  and  $M_u$ , three assumptions are required: (i)  $M_r, M_u$  are complete, deterministic, initially connected and minimal FSMs built a priori (e.g., using automata learning [62]), and (ii) their respective configurations  $\rho_r, \rho_u$ , and (iii) their feature model are known a priori. Thus, we leverage the comparison of state-based models to conditional states and describe how fresh FFSMs can be learnt from two product FSMs.

**DEFINITION 3.1 (FFSM LEARNT FROM TWO CONFIGURATIONS).** An FFSM learnt from  $\langle M_r, M_u \rangle$  is a tuple  $FF = \langle F, \Lambda, C, c_0, Y, O, \Gamma \rangle$ , where (i)  $F = (p_r \cup p_u)$ , (ii)  $\Lambda = \{\rho_r, \rho_u\}$ , (iii)  $C \subseteq S \times B(F)$  is the set of conditional states satisfying conditions (1) and

$$\forall (a, b) \in KPairs, \exists (c_m, \rho_r | \rho_u) \in C \mid a, b \mapsto c_m, \quad (3)$$

$$\forall s_i \in S_r, (s_i, \cdot) \notin KPairs, \exists (c_r, \rho_r) \in C \mid s_i \mapsto c_r, \quad (4)$$

$$\forall s_j \in S_u, (\cdot, s_j) \notin KPairs, \exists (c_u, \rho_u) \in C \mid s_j \mapsto c_u \quad (5)$$

(iv)  $(c_0, true) \in C$  is the initial conditional state such that  $s_{0_r}, s_{0_u} \mapsto c_0$ , (v)  $Y \subseteq (I_r \cup I_u) \times B(F)$  is a finite set of conditional input symbols, (vi)  $O = (O_r \cup O_u)$  is the finite set of output symbols, and (vii)  $\Gamma \subseteq C \times Y \times O \times C$  is the set of conditional transitions satisfying conditions (1-5) and

$$\forall (a_1, b_1), (a_2, b_2) \in KPairs \mid (a_1, x) \in D_r \wedge (b_1, x) \in D_u \quad (6)$$

$$\wedge \lambda_r(a_1, x) = \lambda_u(b_1, x) = o \quad (7)$$

$$\wedge \delta_r(a_1, x) = a_2 \wedge \delta_u(b_1, x) = b_2 \quad (8)$$

$$\exists ((c, \phi), (x, \phi'), o, (c', \phi'')) \in \Gamma \text{ where} \quad (9)$$

$$a_1, b_1 \mapsto c \wedge \phi' = (\rho_u | \rho_r) \wedge a_2, b_2 \mapsto c' \quad (10)$$

to guarantee that transitions of  $M_r$  and  $M_u$  that match origin, input, output and destination (6-8) are combined into one conditional transition (9) annotated with the disjunction of their configurations (10); otherwise, for each defined transition of  $M_r$  and  $M_u$  (11,14), there is a conditional transition (12,15) annotated with their configurations (13,16) as follows:

$$\forall (a_1, x) \in D_r \mid \lambda_r(a_1, x) = o_r \wedge \delta_r(a_1, x) = a_2 \quad (11)$$

$$\exists ((c_r, \phi), (x, \phi'), o_r, (c'_r, \phi'')) \in \Gamma \text{ where} \quad (12)$$

$$a_1 \mapsto c_r \wedge \phi'_r = \rho_r \wedge a_2 \mapsto c'_r \quad (13)$$

$$\forall (b_1, y) \in D_u \mid \lambda_u(b_1, y) = o_u \wedge \delta_u(b_1, y) = b_2 \quad (14)$$

$$\exists ((c_u, \phi), (y, \phi'), o_u, (c'_u, \phi'')) \in \Gamma \text{ where} \quad (15)$$

$$b_1 \mapsto c_u \wedge \phi'_u = \rho_u \wedge b_2 \mapsto c'_u \quad (16)$$

To identify common states, the  $FFSM_{Diff}$  algorithm uses a mapping  $\mapsto$  between states to conditional states of the learnt FFSM. This function ensures that the learnt initial conditional state maps to both initial states, i.e.,  $s_{0_r}, s_{0_u} \mapsto c_0$ . Added to this, we replace the lines 4-6 from Algorithm 1 by  $KPairs \leftarrow (s_{0_r}, s_{0_u}) \cup KPairs$ . Reduce the complexity of feature constraints, states and transitions are annotated with simplified configurations where core features [11] are discarded from their associated formulae.

**Example 3.1 (FFSM learnt from two product configurations).** In Figure 5, we depict a fragment of the FFSM resulting from the comparison of the FSMs in Figures 2 and 4. In this example, the states Pong Game and Bowling Game were merged into one state Bowling\*Pong where there is one conditional transition with input symbol Exit for each configuration. The constraint  $(W \wedge \neg S \wedge \neg B \wedge \neg N)$  is an example of simplified configuration for the product in Figure 2.

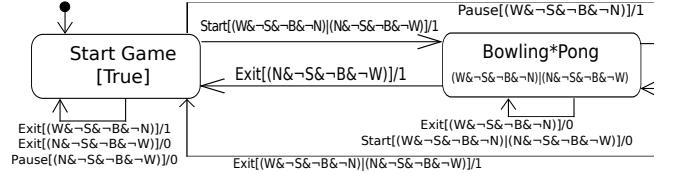


Figure 5: Fragment of the FFSM learnt from the AGM SPL

### 3.2 Including new product behavior into an existing FFSM

Consider the FFSM  $FF_r = \langle F_r, \Lambda_r, C_r, c_{0_r}, Y_r, O_r, \Gamma_r \rangle$  built from a set of configurations  $\Lambda_r$ . If an FFSM  $FF_r$  does not include the behavior of an FSM  $M_u = \langle S_u, s_{0_u}, I_u, O_u, D_u, \delta_u, \lambda_u \rangle$  specifying a configuration  $\rho_u \notin \Lambda_r$ , a new FFSM  $FF$  can be learnt by comparing and merging  $\langle FF_r, M_u \rangle$  using the following definition.

To include a new product into an existing FFSM, three assumptions are required: (iv)  $FF_r, M_u$  are complete, deterministic, initially connected, and minimal built a priori, and (v) configurations  $\rho_u$  is known in advance, and (vi) the FSM and FFSM under learning share a feature model that is known a priori. Thus, on top of Definition 3.1, we define how an FFSM can be enriched with novel behavior.

**DEFINITION 3.2 (FFSM LEARNT FROM  $FF_r$  AND CONFIGURATION  $\rho_u$ ).** An FFSM learnt from  $\langle FF_r, M_u \rangle$  is a tuple  $FF = \langle F, \Lambda, C, c_0, Y, O, \Gamma \rangle$ , where (i)  $F = F_r \cup \{\rho_u\}$ , (ii)  $\Lambda = \Lambda_r \cup \{\rho_u\}$ , (iii)  $C \subseteq S_r \times B(F)$  is the set of states satisfying conditions (1) and

$$\forall (a, b) \in KPairs, \exists (a, \phi_a) \in C_r \wedge (c_m, \phi_a | \rho_u) \in C \mid a, b \mapsto c_m, \quad (17)$$

$$\forall (s_i, \phi_i) \in C_r, (s_i, \cdot) \notin KPairs, \exists (c_r, \phi_i) \in C \mid s_i \mapsto c_r, \quad (18)$$

$$\forall s_j \in S_u, (\cdot, s_j) \notin KPairs, \exists (c_u, \rho_u) \in C \mid s_j \mapsto c_u \quad (19)$$

(iv)  $(c_0, true) \in C$  is the initial conditional state such that the pair  $c_{0_r}, s_{0_u} \mapsto c_0$ , (v)  $Y \subseteq Y_r \cup I_u \times B(F)$  is a finite set of conditional input symbols, (vi)  $O = (O_r \cup O_u)$  is the finite set of output symbols, and (vii)  $\Gamma \subseteq C \times Y \times O \times C$  is the set of conditional transitions satisfying conditions (1), (2), (17-19) and

$$\forall (a_1, b_1), (a_2, b_2) \in KPairs \mid (b_1, x) \in D_u \quad (20)$$

$$\wedge \lambda_u(b_1, x) = o \wedge \delta_u(b_1, x) = b_2 \quad (21)$$

$$\wedge ((a_1, \phi_1), (x, \phi_r), o, (a_2, \phi_2)) \in \Gamma_r \quad (22)$$

$$\exists ((c, \phi), (x, \phi'), o, (c', \phi'')) \in \Gamma \text{ where} \quad (23)$$

$$a_1, b_1 \mapsto c \wedge \phi'_u = (\phi_r | \rho_u) \wedge a_2, b_2 \mapsto c' \quad (24)$$

to guarantee that transitions of  $FF_r$  and  $M_u$  that match origin, input, output and destination (20-22) are combined into conditional transitions (23) annotated with the disjunction of their constraint and configuration (24); otherwise, for each defined transition of  $FF_r$  and  $M_u$  (25,28), there is one conditional transitions (26,29) annotated with their configuration/constraint (27,30) as follows:

$$\forall((a_1, \phi_{a_1}), (x, \phi_r), o_r, (a_2, \phi_{a_2})) \in \Gamma_r \quad (25)$$

$$\exists((c_r, \phi), (x, \phi'_r), o_r, (c'_r, \phi'')) \in \Gamma \text{ where} \quad (26)$$

$$a_1 \mapsto c_r \wedge \phi'_r = \phi_r \wedge a_2 \mapsto c'_r \quad (27)$$

$$\forall(b_1, y) \in D_u | \lambda_u(b_1, y) = o_u \wedge \delta_u(b_1, y) = b_2 \quad (28)$$

$$\exists((c_u, \phi), (y, \phi'_u), o_u, (c'_u, \phi'')) \in \Gamma \text{ where} \quad (29)$$

$$b_1 \mapsto c_u \wedge \phi'_u = \rho_u \wedge b_2 \mapsto c'_u \quad (30)$$

To include a new product into an existing FFSM, lines 4–6 are replaced by  $KPairs \leftarrow (c_{0_r}, s_{0_u}) \cup KPairs$ . Thus, we guarantee that the initial states of  $M_u$  and  $FF_r$  map to the same learnt initial state.

## 4 EMPIRICAL EVALUATION

A family-based analysis technique is effective when its complexity is determined by the number and size of features and the amount of reuse among configurations, rather than the number of valid configurations [58]. Thus, for our technique to qualify as an effective family-based learning technique, we expect to learn *succinct* FFSMs where states and transitions are annotated with *simplified* product configurations. By succinct, we mean that the FFSMs learnt have far fewer states than the total number of states in all products under learning, especially if there is high feature sharing. By simplified, we mean that product configurations are modified by removing core features found using SAT solvers [42].

### 4.1 Research questions

To evaluate the  $FFSM_{Diff}$ , we defined three research questions (RQ). In Table 2, we present our hypotheses about the RQs.

**Table 2: Hypotheses**

RQ	Hypotheses	Description
<b>RQ1</b>	$H_0^{RQ1}$	The size of learnt FFSMs is larger than the total size of the products analyzed
	$H_1^{RQ1}$	The size of learnt FFSMs is at most equal to the total size of the products analyzed
<b>RQ2</b>	$H_0^{RQ2}$	The size of learnt FFSMs is not influenced by the amount of feature sharing
	$H_1^{RQ2}$	The size of learnt FFSMs is influenced by the amount of feature sharing
<b>RQ3</b>	$H_0^{RQ3}$	The learnt FFSMs are larger than the hand-crafted FFSMs
	$H_1^{RQ3}$	The learnt FFSMs have at most the same size as hand-crafted FFSMs

We implemented a tool to compare and combine product FSMs into an FFSM using our algorithm explained in Section 3 and, for feature model analysis, we used the FeatureIDE [59] library and the SAT4J solver [42]. To solve the system of linear equations, we used the Apache Commons Mathematics Library [5]. Details about the experiment design, subject systems and experimental and coding artifacts are presented in Sections 4.2, 4.3 and 4.4, respectively.

As a measure of succinctness, we use the average size of the learnt FFSMs compared to the total size of the products under learning (RQ1) and the size of the hand-crafted FFSMs (RQ3). We describe size in terms of *number of states* as it is one of the factors that influences the complexity of model-based techniques [10, 16, 62]. To measure the *statistical significance*, we used the Mann-Whitney test to check if there was significant difference ( $p < 0.01$ ) between the sizes of the learnt FFSMs and products under learning.

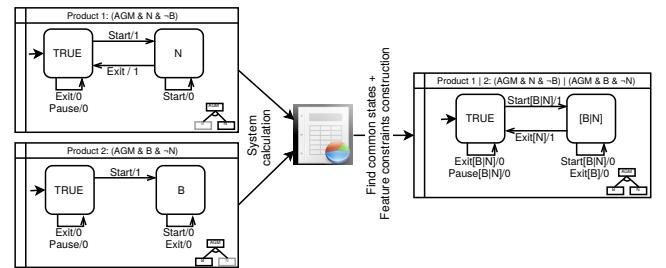
To measure the *scientific significance* [38], we used the Vargha-Delaney's  $\hat{A}$  effect size [64] to assess the probability of the learnt FFSM being more succinct [7]. If  $\hat{A} < 0.5$ , then the learnt FFSM is smaller than the total size of the FSMs of products under learning. If  $\hat{A} = 0.5$ , they have equivalent sizes. To categorize the magnitude of  $\hat{A}$ , we used the intervals between  $\hat{A}$  and 0.5 [33, 60]: *negligible*  $< 0.147 \leq \text{small} < 0.33 \leq \text{medium} < 0.474 \leq \text{large}$ .

Finally, we used Pearson's correlation coefficient to evaluate the relationship between succinctness and reuse (RQ2). Thus, we measured the correlation between the ratio of the size of learnt FFSM to the total size of products analyzed on one hand and the ratio of common features to the total of features implemented by the SULs, on the other hand.

### 4.2 Experiment Design

Let  $\{\rho_0, \rho_1, \dots, \rho_m\} \subseteq B(F)$  be a set of valid configurations, such that the product derived from  $\rho_i$  has at most the same number of states as  $\rho_{i+1}$ , i.e., they are sorted by their FSM size.

For **RQ1** and **RQ2**, we ran the  $FFSM_{Diff}$  for all pairs of products to find common states by solving the system of linear equations and measured the size of the learnt FFSMs. Then, we checked if there were significant and relevant differences between the size of learnt FFSMs and the products analyzed and the correlation between the size of FFSMs and feature reuse. In Figure 6, we illustrate the experiment design.



**Figure 6: Experiment design**

For **RQ3**, added to the FFSMs learnt from pairs of products, we incrementally built FFSMs by merging the FSMs resulting from all products  $\bigcup_{i=0}^{j-1} (\rho_i)$  with the FSM of the next product  $\rho_j$ , and compared the sizes of the learnt and hand-crafted FFSMs.

### 4.3 Subject systems

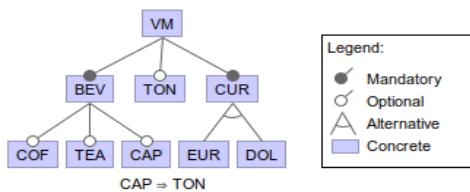
In our evaluation, we used 34 FSMs derived from three SPLs of previous studies [19, 29]. Table 3 depicts the SPLs in terms of the number of features, valid configurations, size of FFSMs and the total sum of the number of states in all valid products analyzed. The AGM SPL is an example from [29] and the Vending Machine (VM) and the Wiper System (WS) are FFSMs hand-crafted by the authors based on FTS [20] from a collection of examples [19].

**Table 3: Description of the SPLs under learning**

SPL		Number of		Sum total of states in	
ID	Name	Features	Valid config.	FFSM	All products
AGM	Arcade Game Maker	13	6	6	21
VM	Vending Machine	9	20	14	207
WS	Wiper System	8	8	13	56

All the FSMs used in this study were derived from FFSMs using the model derivation operator  $\Delta_p$  [29]. Thus, we had reference FSMs and FFSMs to assess the learnt models, namely, the product-line FFSMs were handcrafted and the intermediate FFSMs and the product FSMs were automatically generated. The VM and WS SPLs are described in the following sections.

**4.3.1 Vending Machine.** The Vending Machine (VM) is an SPL that we hand-crafted based on featured transition systems from a collection of illustrative SPLs [19]. In Figure 7, we depict the VM feature model.



**Figure 7: The VM feature model**

In our VM SPL, we support three beverages, (i.e., *Coffee*, *Tea*, and *Cappuccino*), one optional *RingTone* played when the beverage is completed, and two alternative currencies (i.e., *Dollar* or *Euro*). These features composed interesting case as they resulted on FSMs with distinct structures and languages. Among the FSMs derived, we highlight two main differences: (i) the addition of states for each of beverage; (ii) changes in the initial state for each currency. This is our largest SPL in terms of number of products and states.

**4.3.2 Wiper System.** The Wiper System (WS) is another SPL that we hand-crafted based on the aforementioned collection of examples [19]. In Figure 8, we depict the feature model of the WS SPL.

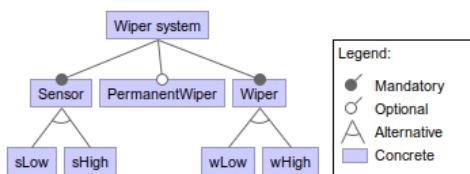


Figure 8: The WS feature model

Our WS SPL has two subsystems – a sensor to detect rain, and the wiper itself; both features are available in two qualities, i.e., *high* and *low*, and one optional feature for permanent movement. A high quality sensor can discriminate between heavy and light rain, whereas a low quality sensor can only distinguish between rain and no rain. Similarly, the high quality wipers can operate at two speeds, and the low quality wiper operates at one single speed.

#### 4.4 Experiment and coding artifacts

For the sake of reproducibility, we have included a lab package with a variety of artifacts (e.g., source-code, test scripts, FFSMs, FSMs, feature models). The lab package is available on GitHub at <https://github.com/damascenodiego/learningFFSM>.

In our lab package, we have included the subject systems, i.e., agm, vm, and ws; and their respective models (i.e., FSMs, FFSMs), feature models, visual representations and test scripts. For the analysis of results, we have included an RStudio [49] project with R scripts for calculating and plotting statistics. To reproduce our experiments, we have included two sets of Python scripts, i.e., `run_<ID>_pairs.py` and `run_<ID>.py`, that execute our tool for each SPL and construct FFSMs from all pairs of products and incrementally learn an FFSM by merging all products.

The source-code of our *FFSM<sub>Diff</sub>* implementation is organized as a Java project where two packages are included: i.e., br.usp.icmc and uk.le.ac. The former includes code artifacts developed by Fragal et al. [29] that we used in the first phase of our study to validate FFSMs and derive FSMs. The latter includes code artifacts that we developed to (i) read/write FSMs; (ii) solve the systems of linear equations to compare the FSMs and FFSMs under learning; and (iii) merge FSM models into annotated FFSMs. The project can be opened using the Eclipse IDE [36] and compiled using the JDK version 1.8.

For reading and writing FSMs, we designed the `ProductMealy` class using the `CompactMealy` class from LearnLib [48]. This class extends basic operations over FSMs (e.g., reset, transition/output functions) with methods to maintain product configurations, as specified by our `IConfigurableFSM` interface.

Using the Apache Commons Math library [5], we calculate the state pairs most likely to be equivalent. Based on empirical observations by [69], we have set the attenuation ratio to  $k = 0.5$  and, to guarantee the mapping between initial states, we have implemented the *identifyLandmarks()* function to return the pair of initial states  $(s_{0_r}, s_{0_u})$  to *KPairs*.

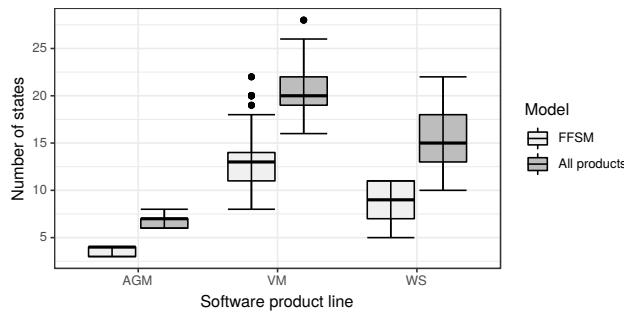
To represent FFSMs, we designed the `FeaturedMealy` class using the `FastNFA` class, one of the LearnLib building blocks to represent non-deterministic models. We opted for a non-deterministic representation because, if we ignore presence conditions, FFSMs can be seen as a non-deterministic FSM. To represent conditional states/transitions, we designed the classes `ConditionalState` and `ConditionalTransition` with collections of `Node` objects. The `Node` class is a `FeatureIDE` building block to represent feature constraints.

## 4.5 Analysis of Results

In this section, we discuss the results of our experiments in terms of the RQs we defined to this study and the Hypotheses we have shown in Table 2.

**4.5.1 Is the FFSM<sub>Diff</sub> algorithm effective in learning succinct family models compared to the total size of the products?**

In Figure 9, we show the boxplots for the size of learnt FFSMs and total size of all products analyzed. The boxplots indicate that all FFSMs learnt from AGM and WS presented fewer states than the average total size of products. For the VM SPL, on the other hand, there were cases where the learnt FFSMs had more states than the hand-crafted FFSMs.



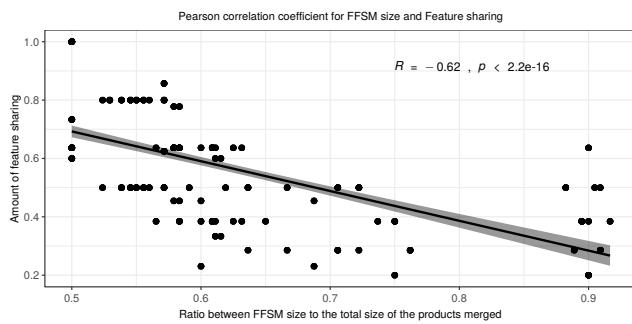
**Figure 9: Boxplots of the size of the learnt FFSMs (white) compared to the total size of all products analyzed (gray)**

We analyzed the FSMs from the VM SPL and found that modifications in the input/output symbols of early transitions and the addition of extra states lead to significant changes in their structure and language. These changes related to features about the currency and drinks supported by the vending machine, e.g., *Dollar*, *Euro*, and played a key role in outgoing transitions from initial states. As result, these changes masqueraded the similarity of surrounding states and hence difficulted their combination. The outliers above the VM boxplot depict these unsuccessful comparisons.

By analyzing the Mann-Whitney test, we found statistically significant differences ( $p < 0.01$ ) between the size of learnt FFSMs and total size of products. The effect size also indicated differences of *large* magnitude with the size of learnt FFSMs as smaller than the total size of all products analyzed. Thus, our results support the hypothesis  $H_1^{RQ1}$  that the size of learnt FFSMs is at most equal to the total size of products analyzed.

#### 4.5.2 Is the size of learnt family models influenced by the amount of feature reuse?

In Figure 10, we show the scatter plot for the amount of feature sharing and size of learnt FFSMs for all pairs of products. We have normalized *the size of learnt FFSMs* using the ratio between the number of states of the learnt FFSM to the total number of states of the products analyzed and *the amount of feature sharing* as the ratio of common features to the total of features.



**Figure 10: Pearson correlation coefficient between FFSM size and amount of feature sharing**

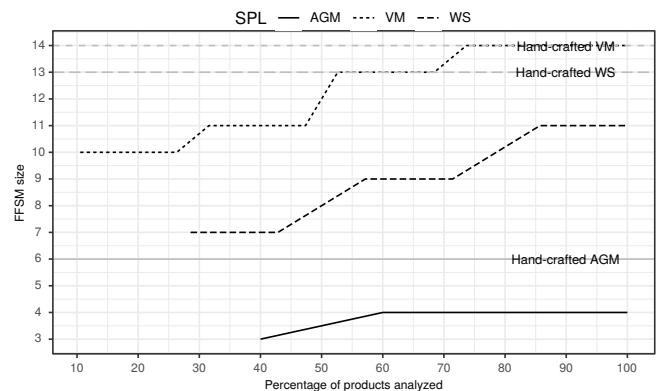
An amount of feature sharing equal to 1.0 means that both products have the same feature configuration. A ratio between size of

learnt FFSM and total size of products equal to 0.5 means that the products analyzed implement equivalent FSMs, otherwise the learnt FFSM includes states depicting variability.

By calculating the Pearson correlation coefficient, we found a strong negative correlation between FFSM size and amount of feature sharing. Thus, FFSMs learnt from products implementing a similar set of features tend to be more succinct than those built from products implementing fewer common features. These findings support our hypothesis  $H_1^{RQ2}$  that the size of learnt FFSMs is influenced by the amount of feature sharing.

#### 4.5.3 Is the $FFSM_{Diff}$ algorithm effective in learning succinct family models compared to hand-crafted family models?

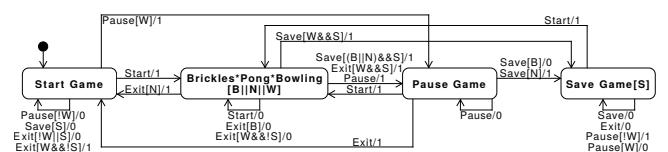
To evaluate the succinctness of the FFSMs learnt using our approach, we also compared the size of hand-crafted models to the FFSMs learnt in two settings: (i) FFSMs incrementally learnt from all products and (ii) FFSMs learnt from pairs of products. In Figure 11, we show the size of learnt FFSMs from all products of the SPLs.



**Figure 11: Size of the FFSMs recovered**

It turns out that two FFSMs were learnt with fewer states than their original models, i.e., AGM and WS. We inspected the learnt and hand-crafted FFSMs models and found that two states presented similar behaviors and could be merged without side-effects. As shown in Table 3, the hand-crafted FFSMs for the AGM and WS SPLs presented 6 and 13 conditional states, respectively. The FFSMs learnt from AGM and WS, in their turn, included 4 and 11 states, respectively.

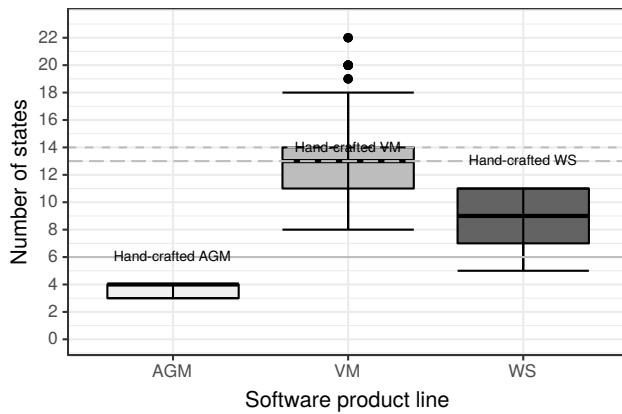
Moreover, we found that the FFSMs learnt from the AGM SPL coincided with the alternative representation in Figure 12. This representation was shown by Fragal [24] in his thesis as an alternative representation to the AGM SPL with fewer states. In this alternative representation, all three conditional states are composed into one state annotated with the disjunction of the alternative features.



**Figure 12: Alternative FFSM for AGM with fewer states [24]**

On the other hand, the hand-crafted and learnt FFSMs for the VM SPL presented the same size. In this case, the order that the products were analyzed favoured the  $FFSM_{Diff}$  algorithm to learn an FFSMs with the same size as the original FFSM of VM SPL. These findings indicate that selecting “good” configurations could be helpful for learning more succinct FFSMs. To achieve this, configuration selection [65] and prioritization [32] can be used to improve the overall t-wise coverage each time a configuration is incorporated into an existing FFSM.

To compare the sizes of FFSMs hand-crafted and learnt from product pairs, we calculated the Mann-Whitney test and  $\hat{A}$  effect size. In Figure 13, we depict boxplots for the sizes of FFSMs learnt from product pairs. The size of hand-crafted FFSMs is shown as gray lines.



**Figure 13: Average size of the learnt FFSMs compared to the size of hand-crafted FFSMs**

By analyzing the results of the Mann-Whitney test, we found statistically significant differences ( $p < 0.01$ ) between the size of FFSMs learnt from AGM and WS and their respective hand-crafted versions. For these SPLs, the effect size indicated differences of *large* magnitude with the learnt FFSMs as the smaller models. For the VM SPL, on the other hand, we did not find statistically significant differences ( $p > 0.01$ ) between the size of the learnt and hand-crafted FFSMs. Thus, our results support the hypothesis  $H_1^{RQ3}$  that learnt FFSMs have at most the same size as hand-crafted FFSMs.

## 4.6 Discussion

In this section, we discuss some of the practical implications and limitations of our study.

**What are the implications for practitioners?** Our work complements traditional reverse engineering techniques on helping practitioners to create family models from product specifications. These specifications can be either created manually or extracted using model learning [62]. As result, our approach can leverage family model-based testing [61] and family model checking [10] to projects where there are no family models a priori.

**What types of models may it work/not work?** In our current implementation, we fixed *identifyLandmarks()* to the mapping between the initial states ( $s_{0_r}, s_{0_u}$ ). As a result, some types of changes

(e.g., those added to initial states of FSMs) hurdled the process of building succinct models. To improve our approach, more parameters can be added to *identifyLandmarks()*, as in its original design where a threshold  $t$  and a ration  $r$  are used to find state pairs most likely to be equivalent [69], or either by allowing engineers to set mappings between state pairs as user-defined assumptions.

### How are the different notions of variability represented?

Currently, our approach annotates state and transitions using the disjunction of simplified configurations. As result, the representation of feature constraints is limited to one single format (i.e., OR with ANDs). To overcome this limitation, more sophisticated presence-condition simplification techniques [67] could be used to reduce the complexity of feature constraints.

## 4.7 Threats to validity

In this section, we discuss the threats to validity of the methods used in this research paper.

**Conclusion validity:** These threats concern the relationship between treatment and outcome. To ensure the reliability of our measures and treatment implementation, we have a setup in place based on widely used tools for automata learning [48], SAT solving [42], and SPL analysis [59].

**External validity:** These concern with the generalization of our results to industrial subject states. Our results are based on three small product lines as subject systems; the small number of product lines and their small sizes pose a threat to external validity. We plan to remedy this by extending our study to more and larger product lines. The fact that the intermediate FFSMs and the product FSMs are generated from the product line FFSM pose another threat to external validity. To remedy this external threat, we need to use learned FSMs along with handcrafted FFSMs from different subsets of larger software product lines. Thus, we could come up with subject systems with more irregularities arising from learning and handcrafting models.

**Internal validity:** These concern the phenomena that can affect the causal relationship between the treatment and outcomes. One variable that will form a threat to internal validity of our results for larger product lines is the *order* of incorporating product FSMs into product-line FFSMs. In our study, we considered one single order for incorporating products into an FFSM. We plan to incorporate this variable into our experimental setup for larger product lines and complement our study with various configuration selection [65] and prioritization [32] techniques.

**Construct validity:** These are concerned with the ability to draw correct conclusions about the treatment and outcomes. Two factors that will form threats to construct validity are the nature of the hand-crafted FFSMs and the engineers expertise. Highly specialized engineers are more likely to come up with optimal models (i.e., minimal number of states) than those with less experience. This may hamper the construction of more succinct models. However, optimal representations are interesting artifacts to show that our technique can also recover models *as succinct as* those hand-crafted by experts.

## 5 RELATED WORK

In this section, we discuss our approach in terms of related work and how it can be helpful in their contexts.

## 5.1 Automata learning

Automata learning [4] has been a popular approach to automatically derive behavioral models. For an overview of related work and an introduction to automata learning and applications, we refer to [2, 37, 56]. Automata learning have been harnessed for black box model checking [46], real-world protocols [1, 23], software evolution [21, 35], automatic test generation [47], and generalization of failure models [17, 40]. The problem of learning models from SPLs becomes more complex as it has to cope with products that may have their own models, requirements and code. Our approach pave the way for family model learning techniques, which are still understudied.

## 5.2 Family-based analysis of SPLs

For an overview on techniques for family-model analysis, testing and modeling, we refer the interested readers to recent surveys [12, 15, 58]. Family models have been exploited as theoretical foundation to perform efficient test case generation [9, 14], family model checking [52, 57], to automatically generate specifications of individual products [8], to support the automatic validation of families of products [29], and to specify fine-grained differences among product variants [53]. Thus, we believe that our approach is complementary to the aforementioned techniques in the sense that it can leverage family model-based techniques to cases where family models are non-existent or outdated.

## 5.3 Comparison of state-based models

The ability to compare FSMs is important for software engineering tasks [69] such as conformance testing [16], and for the sake of evaluating the accuracy of automata learning techniques [62]. Studies related to ours are by Walkinshaw and Bogdanov [69] and Nejati et al. [44].

Walkinshaw and Bogdanov [69] compared two approaches for computing the precise difference between labeled transition systems (LTS) in terms of their language and structure. For comparing the language of state-based models, the authors proposed an approach based on the proportion of test sequences generated by a well-known FSM-based testing method, called the W-method [18, 66], that are classified in the same way by two models  $M_r$  and  $M_u$ . Thus, measurements such as, precision, recall, and F-measure can be used to compare the language of two LTS.

A major issue of comparing FSMs by their language is the fact that minor differences can mask structural similarities. To solve this problem, they proposed the  $LTS_{Diff}$  algorithm, presented in Section 2.3.2. These two approaches are said to be *complementary* as two models may have similar state transition structure, but completely different languages, or vice-versa. In the setting of SPLs, the FSMs specifying products can be compared using the  $LTS_{Diff}$  algorithm. However, the algorithm lacks a step to incorporate feature constraints. Thus, we designed  $FFSM_{Diff}$  to fill this gap.

Nejati et al. [44] presented an approach for matching and merging Statecharts [30]. Their approach relies on two operators for matching and merging transitions. The latter uses static and behavioral properties to match state pairs. The former produces a combined model in which variant behaviors are parameterized using guards on their transitions where temporal properties are preserved. The authors showed that relying on both operators produces higher precision than relying on them independently.

In our study, we aimed at the problem of matching and merging FSMs to build FFSMs. Recently, the FFSM formalism has been extended to support hierarchy [25]. The hierarchical featured finite state machine (HFSM) model improves FFSM readability by grouping up FFSM conditional states and transitions into abstracted entities [25]. The results indicate that HFSMs can be used to succinctly represent and efficiently validate the behavior of parallel components in SPLs. The problem of analyzing Statecharts and learning HFSMs provides interesting possibilities to extend our approach.

## 5.4 Reverse engineering feature models

Feature models play a central role in the variability management for SPLs [11]. Unfortunately, companies often develop software variants in an unstructured manner and may lack feature models as their construction is time-consuming and error prone [31].

In this context, several approaches have been proposed to automatically build feature models from sets of product configurations [3, 31, 51]. Approaches based on Formal Concept Analysis (FCA) show promising possibilities on reverse engineering feature models as they can detect interdependencies and hierarchies between features [3]. Our proposal aims at one similar problem, which can be described as “reverse engineering” family models from product specifications. In our study, we assume that the feature model is known a priori. However, we believe that our technique can be extended to cope with non-existent feature models and learn family and feature models at once, but the succinctness of the feature constraints may be compromised. Thus, investigations combining feature model and behavioral model learning is required.

## 5.5 Software product line evolution

In the context of SPLs, the tasks of re-engineering and refactoring are vital to the maintenance and evolution of their software products. For an overview on SPL evolution, refactoring and reengineering, we refer the interested readers to [22, 41, 43].

A large variety of artifacts have been considered in SPL evolution, but feature models are by far the most researched ones [43]. Moreover, recent studies have shown that there is a need for reengineering approaches specifically tailored for agile processes [43], and migration of SPL paradigms [41].

Several studies have investigated model learning techniques to cope with traditional software evolution and regression testing [34, 55]. However, to the best of our knowledge, there are no works investigating model learning in the setting of SPLs. Combined with automata learning techniques [4], we believe that our algorithm can support model-based regression testing in SPLs [50] and family model checking [52, 57] in agile processes [45].

## 6 CONCLUSION

The problem of outdated and deprecated models can arise in the setting of SPLs and hamper the application of family-based analysis. Family-based analysis techniques operate on a single artifact, referred to as a family model, annotated with feature constraints to express variability in terms of states and transitions specific to product configurations. To tackle this issue, we introduce  $FFSM_{Diff}$ , an automated technique to learn succinct FFSMs from sets of FSMs specifying products.

Our technique incorporates variability to compare and merge FFSMs and annotate states and transitions with feature constraints. We designed our approach to cope with (i) learning fresh FFSMs from two FSMs, and (ii) including an FSM into an existing FFSM. To evaluate our technique, we used 34 products derived from three SPLs and measured its effectiveness in terms of the size of learnt FFSMs and amount of feature reuse.

Our results supported the hypothesis that families of FSMs can be effectively merged into succinct FFSMs, especially if there is high feature reuse among products. These indicate that  $FFSM_{Diff}$  is an efficient family-based model learning technique that can pave the way to several family-based analysis techniques without family models specified *a priori*.

As future work, we would like to investigate further how family models can be used to steer the process model learning. Adaptive learning is a variant of model learning that attempts to reuse sequences from existing models to speed up state discovery [34]. We believe that FFSMs can be useful to derive queries and improve the process of reverse engineering family models. Another possible branch of this research consist of evaluating how configuration prioritization [32] may affect the size of the family models. Our results indicated that some combinations may lead to FFSMs larger than hand-crafted versions and we believe that similarity functions [32] may be useful to accelerate and support family model learning. The problem of learning HFSMs from Statecharts is left as future work.

## ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and University of Leicester, College of Science & Engineering. Research carried out using the computational resources of the Center for Mathematical Sciences Applied to Industry (CeMEAI) funded by FAPESP (grant 2013/07375-0). The authors are also grateful to the anonymous reviewers; the *VALidation and Verification* (VALVE) research group at the University of Leicester; and the *Laboratory of Software Engineering* (LabES) at the University of São Paulo (ICMC-USP) for their useful comments and suggestions.

## REFERENCES

- [1] Fides Aarts, Faranak Heidian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. 2012. *Automata Learning through Counterexample Guided Abstraction Refinement*. Springer, Berlin, Heidelberg, 10–27.
- [2] Bernhard K. Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappeler, and Masoumeh Taromirad. 2018. Model Learning and Model-Based Testing. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24–27, 2016, Revised Papers*, Amel Bennaceur, Reiner Hähnle, and Karl Meinke (Eds.). Springer International Publishing, Cham, 74–100.
- [3] Ra'fat Ahmad Al-Msie'Deen, Marianne Huchard, Abdelhak-Djamel Seriai, Christelle Urtado, and Sylvain Vaughtier. 2014. Reverse Engineering Feature Models from Software Configurations using Formal Concept Analysis. In *CLA: Concept Lattices and their Applications*, Karel Bertet and Sebastian Rudolph (Eds.), Vol. 1252, 95–106.
- [4] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (1987), 87–106.
- [5] Apache. 2016. Commons Math: The Apache Commons Mathematics Library. <http://commons.apache.org/proper/commons-math/>. [Online; accessed 28-Mar-2019].
- [6] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*. ACM, New York, NY, USA, 1–8. <https://doi.org/10.1145/2528265.2528267>
- [7] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 1–10.
- [8] Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, and Stefania Gnesi. 2012. *A Compositional Framework to Derive Product Line Behavioural Descriptions*. Springer, Berlin, Heidelberg, 146–161.
- [9] Joanne M. Atlee, Sandy Beidu, Uli Fahrenberg, and Axel Legay. 2015. Merging Features in Featured Transition Systems. In *Proceedings of the 12th Workshop on Model-Driven Engineering, Verification and Validation co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (CEUR Workshop Proceedings)*, Vol. 1514, 38–43.
- [10] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press.
- [11] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–636. <https://doi.org/10.1016/j.is.2010.01.001>
- [12] Fabian Benduhn, Thomas Thüm, Malte Lochau, Thomas Leich, and Gunter Saake. 2015. A Survey on Modeling Techniques for Formal Behavioral Verification of Software Product Lines. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '15)*. ACM, New York, NY, USA, 80–87.
- [13] Harsh Beohar and Mohammad Reza Mousavi. 2014. Input-output conformance testing based on featured transition systems. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14*. ACM Press, New York, New York, USA, 1272–1278. <https://doi.org/10.1145/2554850.2554949>
- [14] Harsh Beohar and Mohammad Reza Mousavi. 2016. Input-Output Conformance Testing for Software Product Lines. *Journal of Logical and Algebraic Methods in Programming* 85, 6 (2016), 1131–1153.
- [15] Harsh Beohar, Mahsa Varshosaz, and Mohammad Reza Mousavi. 2016. Basic behavioral models for software product lines: Expressiveness and testing pre-orders. *Science of Computer Programming* 123 (2016), 42–60.
- [16] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. 2005. *Model-Based Testing of Reactive Systems: Advanced Lectures*. Springer, Berlin, Heidelberg, 557–603 pages.
- [17] Martin Chapman, Hana Chockler, Pascal Kesseli, Daniel Kroening, Ofer Strichman, and Michael Tautschig. 2015. *Learning the Language of Error*. Springer International Publishing, Cham, 114–130.
- [18] T. S. Chow. 1978. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering* 4, 3 (May 1978), 178–187.
- [19] Andreas Classen. 2010. Modelling with FTS: a Collection of Illustrative Examples. <https://researchportal.unamur.be/en/publications/modelling-with-fts-a-collection-of-illustrative-examples>
- [20] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering* 39, 8 (Aug 2013), 1069–1089.
- [21] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC '15)*. USENIX Association, Berkeley, CA, USA, 193–206.
- [22] Wolfram Fenske, Thomas Thüm, and Gunter Saake. 2013. A Taxonomy of Software Product Line Reengineering. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '14)*. ACM, New York, NY, USA, 1–8.
- [23] Paul Fiterău-Broştean and Falk Howar. 2017. *Learning-Based Testing the Sliding Window Behavior of TCP Implementations*. Springer International Publishing, Cham, 185–200.
- [24] Vanderson Hafemann Fragal. 2017. *Automatic generation of configurable test-suites for software product lines*. Ph.D. Dissertation, Universidade de São Paulo. [Online] <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-10012019-085746>.
- [25] Vanderson Hafemann Fragal, Adenilso Simao, and Mohammad Reza Mousavi. 2019. Hierarchical featured state machines. *Science of Computer Programming* 171 (2019), 67–88.
- [26] Vanderson Hafemann Fragal, Adenilso Simao, Mohammad Reza Mousavi, and Uraz Cengiz Turker. 2018. Extending HSI Test Generation Method for Software Product Lines. *Comput. J.* 62, 1 (05 2018), 109–129.
- [27] Arthur Gill. 1962. *Introduction to the Theory of Finite State Machines*. McGraw-Hill, New York.
- [28] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. 2008. *Modeling and Model Checking Software Product Lines*. Springer, Berlin, Heidelberg, 113–131.
- [29] Vanderson Hafemann Fragal, Adenilso Simao, and Mohammad Reza Mousavi. 2017. *Validated Test Models for Software Product Lines: Featured Finite State Machines*. Springer International Publishing, Cham, 210–227.
- [30] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8, 3 (June 1987), 231–274.

- [31] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrezon, and Alexander Egyed. 2011. Reverse Engineering Feature Models from Programs' Feature Sets. In *2011 18th Working Conference on Reverse Engineering*. IEEE, 308–312.
- [32] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering* 40, 7 (2014), 650–670.
- [33] Melinda R Hess and Jeffrey D Kromrey. 2004. Robust Confidence Intervals for Effect Sizes: A Comparative Study of Cohen's d and Cliff's Delta Under Non-normality and Heterogeneous Variances. In *Annual meeting of the American Educational Research Association*.
- [34] David Huistra, Jeroen Meijer, and Jaco van de Pol. 2018. Adaptive Learning for Learn-Based Regression Testing. In *Formal Methods for Industrial Critical Systems (Lecture Notes in Computer Science)*, Falk Howar and Jiri Barnat (Eds.). Springer Publishers, Switzerland, 162–177.
- [35] Hardi Hungar, Oliver Niese, and Bernhard Steffen. 2003. *Domain-Specific Optimization in Automata Learning*. Springer, Berlin, Heidelberg, 315–327.
- [36] Eclipse IDE. 2019. Eclipse desktop and Web IDEs. <https://www.eclipse.org/ide/>. [Online; accessed 19-May-19].
- [37] Muhammad Naeem Irfan, Catherine Oriat, and Roland Groz. 2013. Chapter 3 - Model Inference and Testing. *Advances in Computers*, Vol. 89. Elsevier, 89–139.
- [38] Vigdis By Kampenes, Tore Dyba, Jo E. Hannay, and Dag I.K. Sjoberg. 2007. A systematic review of effect size in software engineering experiments. *Information and Software Technology* 49, 11 (2007), 1073–1086.
- [39] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
- [40] Sebastian Kunze, Wojciech Mostowski, Mohammad Reza Mousavi, and Mahsa Varshosaz. 2016. Generation of Failure Models through Automata Learning. In *2016 Workshop on Automotive Systems/Software Architectures (WASA)*. 22–25.
- [41] Miguel A. Laguna and Yania Crespo. 2013. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Science of Computer Programming* 78, 8 (2013), 1010–1034.
- [42] Daniel Le Berre and Anne Parrain. 2010. The SAT4J library, Release 2.2, System Description. *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), 59–64.
- [43] MaÃ±ra Marques, Jocelyn Simmonds, Pedro O. Rossel, and MaÃ±a Cecilia Basartica. 2019. Software product line evolution: A systematic literature review. *Information and Software Technology* 105 (2019), 190–208.
- [44] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. 2012. Matching and Merging of Variant Feature Specifications. *IEEE Transactions on Software Engineering* 38, 6 (Nov 2012), 1355–1375.
- [45] Johannes Neubauer, Bernhard Steffen, Oliver Bauer, Stephan Windmller, Maik Merten, Tiziana Margaria, and Falk Howar. 2012. Automated continuous quality assurance. In *2012 First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*. IEEE, 37–43.
- [46] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. 1999. *Black Box Checking*. Springer US, Boston, MA, 225–240.
- [47] Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. 2009. Dynamic testing via automata learning. *International Journal on Software Tools for Technology Transfer* 11, 4 (2009), 307.
- [48] Harald Raffelt and Bernhard Steffen. 2006. *LearnLib: A Library for Automata Learning and Experimentation*. Springer, Berlin, Heidelberg, 377–380.
- [49] RStudio. 2019. RStudio: Open source and enterprise-ready professional software for data science. <https://www.rstudio.com/>. [Online; accessed 19-May-19].
- [50] Per Runeson and Engelin Engström. 2012. Chapter 7 - Regression Testing in Software Product Line Engineering. *Advances in Computers*, Vol. 86. Elsevier, 223–263.
- [51] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. 2011. Extraction of Feature Models from Formal Contexts. In *Proceedings of the 15th International Software Product Line Conference, Volume 2 (SPLC '11)*. ACM, New York, NY, USA, 1–8.
- [52] Hamideh Sabouri and Ramtin Khosravi. 2013. *Delta Modeling and Model Checking of Product Families*. Springer, Berlin, Heidelberg, 51–65.
- [53] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanarella. 2010. *Delta-Oriented Programming of Software Product Lines*. Springer, Berlin, Heidelberg, 77–91.
- [54] Ina Schaefer, Rich Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 477–495.
- [55] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. 2015. *Incremental Upgrade Checking*. Springer International Publishing, Cham, 55–72.
- [56] Andrew Stevenson and James R. Cordy. 2014. A Survey of Grammatical Inference in Software Engineering. *Sci. Comput. Program.* 96, P4 (Dec. 2014), 444–459.
- [57] Maurice H. ter Beek, Erik P. de Vink, and Tim A. C. Willemse. 2017. *Family-Based Model Checking with mCRL2*. Springer, Berlin, Heidelberg, 387–405.
- [58] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (June 2014), 1–45.
- [59] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79, Supplement C (2014), 70–85.
- [60] Marco Torchiano. 2017. *effsize: Efficient Effect Size Computation* (v. 0.7.1). CRAN package repository. <https://cran.r-project.org/web/packages/effsize/effsize.pdf> [Online; accessed 20-November-2017].
- [61] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability* 22, 5 (2012), 297–312.
- [62] Frits Vaandrager. 2017. Model Learning. *Commun. ACM* 60, 2 (Jan. 2017), 86–95.
- [63] Tassio Vale, Eduardo Santana de Almeida, Vander Alves, Uirâa Kulesza, Nan Niu, and Ricardo de Lima. 2017. Software product lines traceability: A systematic mapping study. *Information and Software Technology* 84 (2017), 1–18.
- [64] AndrÃas Varga and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.
- [65] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 1 (SPLC '18)*. ACM, New York, NY, USA, 1–13.
- [66] M. P. Vasilevskii. 1973. Failure diagnosis of automata. *Cybernetics* 9, 4 (1973), 653–665.
- [67] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-condition Simplification in Highly Configurable Systems. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE, Piscataway, NJ, USA, 178–188.
- [68] Neil Walkinshaw. 2013. Chapter 1 - Reverse-Engineering Software Behavior. In *Advances in Computers*, Atif Memon (Ed.). Advances in Computers, Vol. 91. Elsevier, 1–58.
- [69] Neil Walkinshaw and Kirill Bogdanov. 2013. Automated Comparison of State-Based Software Models in Terms of Their Language and Structure. *ACM Transactions on Software Engineering and Methodology* 22, 2 (March 2013), 1–37.

# Feature-Family-Based Reliability Analysis of Software Product Lines

Andre Lanna  
Thiago Castro  
Vander Alves  
Genaina Rodrigues  
University of Brasilia  
Brasilia, DF, Brazil

Pierre-Yves Schobbens  
University of Namur  
Namur, Belgium

Sven Apel  
Saarland University  
Saarbrücken, Germany

## ABSTRACT

**Context:** Verification techniques such as model checking are being applied to ensure that software systems achieve desired quality levels and fulfill their functional and non-functional specification. However, applying these techniques to software product lines is a twofold challenge, given the exponential blowup of the number of products and the state-explosion problem inherent to model checking. Current product-line verification techniques leverage symbolic model checking and variability information to optimize the analysis but still face limitations that make them costly or infeasible. In particular, state-of-the-art verification techniques for *product-line reliability analysis* are enumerative which hinders their applicability, given the latent blowup of the configuration space.

**Objective:** Our objectives are the following: (a) we present a method to efficiently compute the reliability of all configurations of a compositional or annotation-based software product line from its UML behavioral models, (b) we provide a tool that implements the proposed method, and (c) we report on an empirical study comparing the performance of different reliability analysis strategies for software product lines.

**Method:** We present a novel *feature-family-based* analysis strategy to compute the reliability of all products of a (compositional or annotation-based) software product line. The strategy employs a divide-and-conquer approach over UML behavioral models endowed with probabilistic and variability information. The *feature-based* step of our strategy *divides* the behavioral models into smaller feature-dependent fragments that can be analyzed more efficiently. Such analysis consists of creating a probabilistic model for each behavioral fragment and analyzing such model using a parametric model checker that returns an expression denoting its reliability. Parameters in such expression represent the reliabilities of fragments on which it depends at runtime. The *family-based* step performs the reliability computation for all configurations at once (*conquer*) by evaluating reliability expressions in terms of a *suitable variational data structure*. This step solves the expression computed for each behavioral fragment taking into account (a) the fragment's variability information and (b) the reliability values already computed

for the fragments on which it depends. The result is an Algebraic Decision Diagram (ADD) whose terminals different than zero represent the reliability value of valid (partial) configurations for the fragment. Therefore, the ADD computed for the last evaluated fragment contains the reliability values for all valid configurations of the software product line.

**Results:** We performed an experiment to compare our feature-family-based and other four state-of-the-art evaluation strategies (product-based, family-based, feature-product-based and family-product-based). The subjects were variations of six publicly available product lines, whose configuration spaces were progressively increased. The empirical results show that our feature-family-based strategy outperforms, in terms of time and space, the other four state-of-the-art strategies. In addition, it is the only one that could be scaled to a  $2^{20}$ -fold increase in the size of the configuration space.

**Conclusion:** Our feature-family-based strategy leverages both feature-based and family-based strategies by taming the size of the models to be analyzed (due to the decomposition of behavioral models into fragments) and by avoiding the products enumeration inherent to some state-of-the-art analysis methods by using ADDs to represent both variability and reliability values.

**Journal paper:** This paper was published at the Information and Software Technology Journal. It is available at <https://doi.org/10.1016/j.infsof.2017.10.001>.

**Supplementary material:** Additional material to the IST submission is available at <https://splmc.github.io/scalabilityAnalysis/>. This material comprises experiments data, the tool implementing the feature-family-based reliability analysis strategy and the environment for experiment replication.

## CCS CONCEPTS

- Software and its engineering → Software product lines; Software reliability; Model checking.

## KEYWORDS

Parametric verification, Software product lines, Software reliability analysis

## ACM Reference Format:

Andre Lanna, Thiago Castro, Vander Alves, Genaina Rodrigues, Pierre-Yves Schobbens, and Sven Apel. 2019. Feature-Family-Based Reliability Analysis of Software Product Lines. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342376>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7138-4/19/09.

<https://doi.org/10.1145/3336294.3342376>

# Variability-Aware Semantic Slicing Using Code Property Graphs

Lea Gerling

University of Hildesheim, Institute of Computer Science  
Hildesheim, Germany  
gerling@sse.uni-hildesheim.de

Klaus Schmid

University of Hildesheim, Institute of Computer Science  
Hildesheim, Germany  
schmid@sse.uni-hildesheim.de

## ABSTRACT

A common problem in program analysis is to identify semantically related statements in programs, for example, which statements change the value of a variable, or implement a specific feature or functionality. This is a very challenging task for large programs and gets even more complicated in the presence of variability implementations like `#ifdef`-annotations. Program slicing is a technique that can be used to aid developers with this challenge. But while slicing is a well-established technique for individual programs, there has been so far only little work on program slicing of product lines.

Here, we introduce a static-analysis approach for semantic slicing of product lines. Our approach introduces the novel concept of a variability-aware code property graph, which combines information about code properties (like statement type) and syntactical structure with data- and control-flow information. This graph is then traversed to gather semantically-related lines of code for a given entry node. We demonstrate our approach with a C-example, including preprocessor statements.

## CCS CONCEPTS

- Software and its engineering → Automated static analysis; Software product lines;
- Information systems → Graph-based database models.

## KEYWORDS

Software Product Lines, Code Property Graphs, Semantic Slicing

### ACM Reference Format:

Lea Gerling and Klaus Schmid. 2019. Variability-Aware Semantic Slicing Using Code Property Graphs. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3336294.3336312>

## 1 INTRODUCTION

Modern software tends to be complex, especially in the presence of `#ifdef`-annotations [8, 19]. Hence, it is very difficult for a developer to understand how one statement is connected with other statements. This leads to several challenges, for example when the developer needs to determine the impact of a change or wants to locate the implementation of a specific functionality and its dependencies, which

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.  
ACM ISBN 978-1-4503-7138-4/19/09...\$15.00  
<https://doi.org/10.1145/3336294.3336312>

are common tasks in the context of software evolution. To help the developer deal with such challenges, there are several approaches that aim at reducing the complexity of a given program. One important technique is *program slicing*, which was first introduced by Weiser [21]. Program slicing computes a reduced slice of a program, based on the analysis of control- and data-flow. The slice contains only those lines of code that influence a given slicing criterion, e.g., the value of variable `x` on line 5.

Since the first introduction of program slicing, various slicing techniques with different applications were introduced [18]. Our target application is the *semantic slicing* of software product lines. Semantic slicing aims at identifying the part of the implementation that semantically influences a given location in the program. The resulting slice does not only help developers to better understand the source code by reducing its complexity, it can also provide a basis for patch-generation, when developers want to migrate a specific functionality from one software to another. This is a common use case when systematic product line engineering is combined with clone-and-own approaches [10]. To address this use case, the slicing approach must fulfill the following criteria:

- (C1) The variability-structure and -information in the resulting program slice stays intact
- (C2) Therefore, preprocessor statements must be contained in the data structure that serves as a basis for the slicing approach

In contrast to other static analyses for product lines, like, for example, TypeChef [13], SPLIFT [5] or others [20], we do not aim at analyzing certain characteristics of the software. Instead, our goal is to generate a semantic program slice that can be turned into a patch, comparable to the goal of the work by Li et al. [14]. Thus, the slice must contain the specific source code statements that appear in the implementation as defined in criterion (C1).

Furthermore, slicing approaches rely on a representation of the program that contains control- and data-flow information, typically implemented by a dependence graph. The nodes of this graph describe source code statements. A slicing algorithm calculates which of these nodes can be reached, based on the slicing criterion. Therefore, the slice can only contain the source code statements that are present in the graph. This leads to the definition of criterion (C2). Currently, there exists no approach that fulfills these two criteria. To address this gap, we make the following contributions:

- In Section 3.1, we present the idea of variability-aware code property graphs, a novel graph-based code representation.
- In Section 3.2, we describe our static-analysis-based semantic slicing approach and demonstrate its applicability on variable software with preprocessor annotations.

```
1 /*File: aSndr.c */
2 #include "drvUtils.h"
3 #include "valCmp.h"
4 #include "Sndr.h"
5
6 #ifdef analogueSender
7     int sendASignal(int *signal){
8         int var = utilFunction(&signal);
9         doSomethingImportant(var);
10        return var;
11    }
12 #endif
13
14 /*File: drvUtils.c */
15 int utilFunction(int *i){
16     /* some useful utilities */
17     return i;
18 }
19
20 /*File: cnvrt.c */
21 #ifdef analogueSender
22     int convertASignal(int *signal){
23         /* some converting is done */
24         return signal;
25     }
26 #endif
27
28 /*File: valCmp.h */
29 #ifdef otherFeature
30     #define doSomethingImportant(var) var*2
31 #endif
```

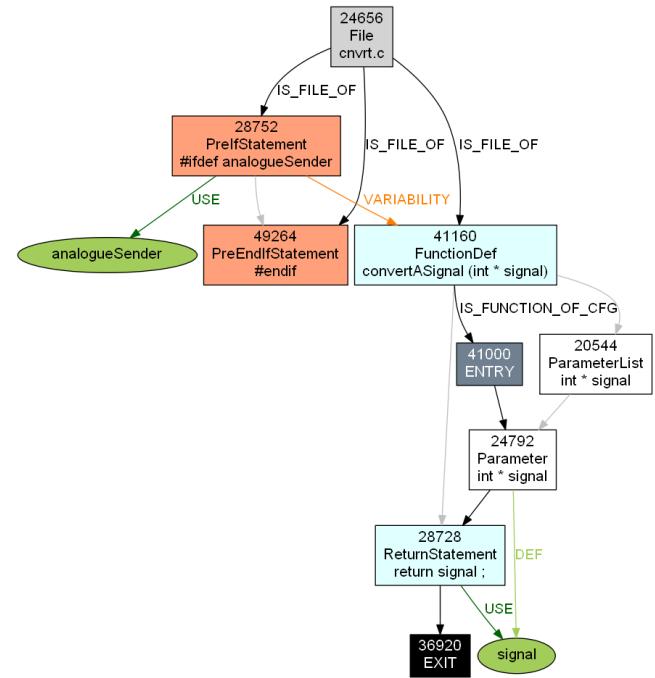
**Listing 1:** A product line example with C preprocessor

## 2 PROBLEM STATEMENT

We use the following example to illustrate the problem: Listing 1 shows an excerpt of a product line, which uses C preprocessor statements to implement variability. Furthermore, there exist various forks of this product line, which use clone-and-own for custom adaptations. The configuration options are scattered over .c and .h files. Listing 1 contains the configuration options `analogueSender` in the files `aSndr.c` and `cnvrt.c`, as well as the configuration option `otherFeature` in `valCmp.h`. The feature-to-code-mapping is not well documented, which is a typical issue in practice.

It is planned to migrate a specific functionality from the main product line to a fork that has been developed separately for several years. It is known that the functionality is connected to the configuration option `analogueSender`. However, it is not known whether there are other statements relevant for the functionality, which are not annotated with this configuration option. In our example, the `utilFunction` in `drvUtls.c` is not surrounded by any `#ifdef`-statements, but is called by the `sendASignal` function in `aSnrd.c`. Furthermore, the `doSomethingImportant` macro in `valCmp.h` changes a value in the `convertASignal` function in `cnvrt.c`, but is annotated with a different configuration option. Nevertheless, the `doSomethingImportant` macro has an important impact on the functionality and therefore should be migrated.

The dependency between `analogueSender` and `otherFeature` may be described somewhere in a variability model, but is hard to find solely from analyzing the code. As a consequence, the described scenario often results in a developer conducting time-consuming manual analysis with lots of trial and error. To solve this problem, there are in principle two different approaches conceivable: The use of search-based approaches, like, for example, feature transplantation [4] or CSlicer [14]; or the use of static-analysis-based techniques,



**Figure 1: An example code property graph**

like program slicing [18]. Search-based approaches are not applicable without major additional effort, because in the context of C software, there are usually no or very few unit test cases. Furthermore, these test cases must be able to cope with preprocessor code.

Existing slicing approaches are not applicable either, because of how the underlying data structure is computed. For example, the well known tool TypeChef [13] parses the C code into an AST and enhances it with variability information, like, for example, conditional nodes that represent a choice. This variability information is gathered by resolving all preprocessor statements during the lexing process. TypeChef and all other approaches that resolve preprocessor statements work well for static analyses like type checking, but they are not sufficient to solve the problem described in this section. A slicing approach based on such graphs without preprocessor statements has the following limitations:

- No preprocessor statements can be part of the slicing criterion, therefore it is not possible to slice for the `analogueSender` configuration option
  - The resulting slice can not contain preprocessor statements, therefore the variability information in Lines 6, 12, 539, 544, 699 and 701, as well as the included files or the macro in line 700 are missing in the result

### 3 SOLUTION APPROACH

In this section, we propose our novel approach for variability-aware semantic slicing using code property graphs that fulfills the criteria **(C1)** and **(C2)**. First, we introduce the concept of variability-aware code property graphs. Second, we provide an overview of the semantic slicing process and illustrate the applicability of our approach using the example from Section 2. Finally, we discuss the current limitations of our approach.

### 3.1 Code Property Graphs

For our semantic slicing approach, we need three different forms of abstractions of the source code: we need information regarding the structure of the source code, which is typically represented by an AST; we require information on the control-flow within the program and, finally, we require information regarding the data-flow within the program. All three of these imply dependency information that is relevant for aggregating a slice, starting from a given entry point. In principle, this process could rely on three separate information structures, however, it is technically easier to rely on a single integrated data structure. Such a data structure has been called *code property graphs* by Yamaguchi et al. [22]. A code property graph is a directed, attributed graph that combines all of these three abstractions into a single graph representation. It consists of nodes that represent source code and edges that represent relationships among code fragments. Each node has various properties, like a unique id, statement type, location in the project, or number of children. The edges are attributed with the relation type, like USE, VARIABILITY, etc. to describe their semantic. One of our contributions is the variability-awareness of this graph, which is realized by parsing preprocessor statements directly into the graph and adding explicit variability links.

Figure 1 shows an example of a variability-aware code property graph. It is based on the source code from `cnvrt.c`, given in Listing 1. The graph lacks some dispensable details, like all AST children, to simplify the representation. The top node is a File node with id 24656 and content `cnvrt.c`, representing the filename. Further, the graph represents the AST of the source code by square nodes that are connected by gray edges. Each edge represents an *is-parent-of* relation. Instead of combining preprocessor and C statements with *is-parent-of* relations, the different languages can only be connected with VARIABILITY links that mark the influence of a configuration option. The light blue squares illustrate the statements that directly appear in the source code. The grey ENTRY node and the black EXIT node mark beginning and end of the control flow analysis. The black arrows show the control flow relations. The round green nodes named `analogueSender` and `signal` illustrate the data flow. They have incoming USE edges when the value of the respective variable or configuration option is read and incoming DEF edges when the value is changed. The `#ifdef analogueSender` node is integrated into the data-flow representation, because preprocessor statements can also use or define C-variables, like the macro in `valCmp.h` in Listing 1.

### 3.2 Semantic Slicing

Our variability-aware semantic slicing approach traverses the code property graph described in Section 3.1. The general slicing process is illustrated in Figure 2. The first step of the process is to select one or more *entry points* in the source code. An entry point marks the starting point for our algorithm. Entry points are manually set by a developer. Our approach will then iteratively gather all statements that are semantically related to the entry point. To achieve this, the algorithm maintains two separate sets of nodes: One *analysis set* and one *result set*. After the process is finished, the latter contains all semantically related statements. The *analysis set* contains all nodes that are scheduled for further analysis. This analysis does not aim at analyzing the node itself, instead the goal is to find the next semantically related nodes to the current node. This step is

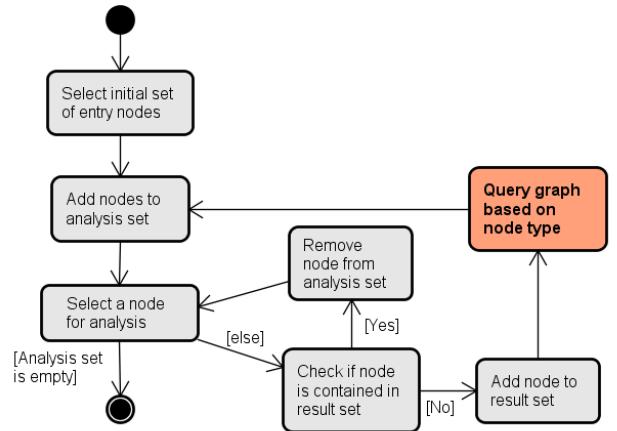


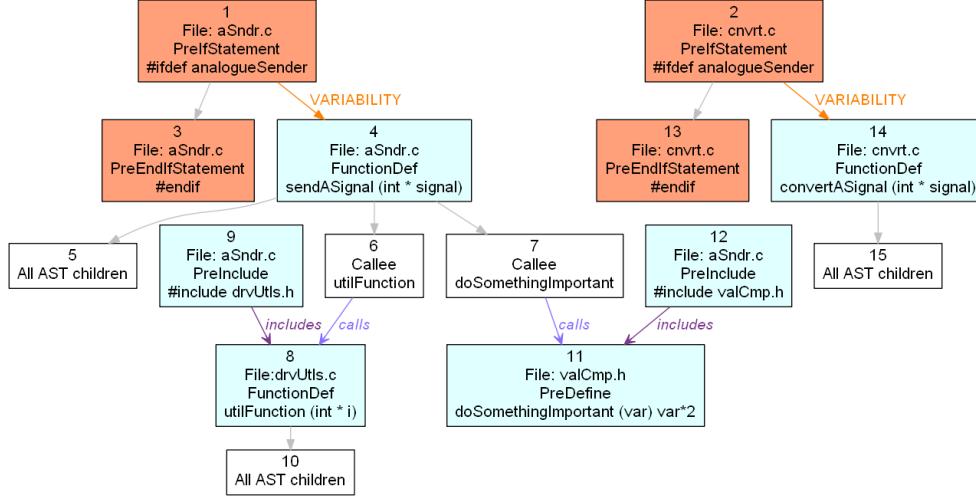
Figure 2: The semantic slicing process

highlighted in orange in Figure 2. Our contribution is the variability- and semantic-awareness of this analysis step as discussed next.

For each node type, which can appear in a code property graph, we created a configurable rule set that describes, which edges are traversed when determining the semantically related nodes. The algorithm can traverse incoming edges as well as outgoing edges, therefore it combines the use of *forward* and *backward-slicing*. These rules can be configured to allow users to adapt the analysis according to their needs. For example, the algorithm can be configured to traverse all incoming *calls* relations to a given function, or to just traverse outgoing *calls* edges from AST children of the function.

Below, we will demonstrate the applicability of our approach based on the example code in Listing 1 and the problem scenario described in Section 2. The whole process is visualized in Figure 3. The top number inside the nodes serves as identifier and also describes the order in which the nodes are added to the *analysis set*. Some nodes contain their location to make the reference to the example code easier. Additionally, there are two new edge types: *includes* and *calls*. They are in this figure to graphically represent two relations that were traversed during the slicing process with special queries. They are not contained in the code property graph, as the computation of these relations is more efficient after the graph is built. Finally, the *All AST children* nodes represent a variable number of AST child nodes of its parent node to simplify the illustration.

In the example for our semantic slicing approach we use the configuration option `analogueSender` as entry point. Therefore, the first step is to identify all preprocessor nodes that refer to the `analogueSender` configuration option and then add them to the *analysis set*. They are identified by searching all conditional compilation node contents for the string `analogueSender`. In our example, only the `PreIfStatements` in file `aSnrd.c` and `cnvrt.c` contain a reference to this configuration option. Next, our approach searches for AST children of Node 1. In our example, the result of this search contains only the `#endif` statement in file `aSnrd.c`. This is the third node added to the *analysis set*. The last step of the analysis of Node 1 is to follow all outgoing *CONTROLS* edges. Here, we add the `sendASignal` function to the *analysis set*.



**Figure 3: A stepwise example, based on Listing 1, for the semantic slicing process with `analogueSender` as entry point**

Node 4 is analyzed next. It is a `FunctionDef` and contains several AST children, which are all added to the `analysis set`. The analysis continues with each AST children of Node 4. Only the two `Callee` nodes trigger further analysis: a search for the called function or macro. For this purpose, the algorithm searches first in the graph database for `FunctionDef` or `PreDefine` nodes that contain the string `utilFunction`. As the `utilFunction` is in another file, which is known from the properties of the respective node, the algorithm searches for `PreInclude` statements in the file of the `Callee` that refer to the header file of the called function. Thus, the algorithm adds the `#include "drvUtils.h"` node to the `analysis set`. Then, all AST children of the `utilFunction` are added to the `analysis set`. After that, the analysis continues with Node 7, using the same procedure as for Node 6.

Finally, the algorithm conducts the analysis of Node 2 and adds Node 13, 14 and 15 to the `analysis set`. The semantic slicing approach stops, when all nodes from the `analysis set` are analyzed. No node or edge is visited twice. In our example, we configured the approach to not follow incoming `VARIABILITY` edges from other configuration options, as we wanted to focus exclusively on the `analogueSender` feature and limit the size of the resulting slice. As a consequence, the preprocessor statements in line 699 and 701 from Listing 1 are not included in the result. If we configured otherwise, the slicing approach would continue to look for all preprocessor statements that include the `otherFeature` configuration option and the resulting slice would be much bigger.

### 3.3 Limitations

In this section, we discuss the limitations of our proposed semantic slicing approach. We distinguish among platform limitations, which are inherited from the reuse of existing tools, conceptual limitations, and limitations of the current implementation.

**Platform limitations:** We reuse the tool Joern [22] for the generation of code property graphs. Joern's data- and control-flow analysis is only intraprocedural. Therefore, some constructs like global variables are currently not completely integrated in the analysis. These

missing explicit edges can in principle be replaced by respective search operations, like demonstrated in Section 3.2 with the `includes` and `calls` relations. Another more general limitation is the C pointer analysis: It is currently not possible to correctly analyze all pointer interactions, therefore we do not claim to be complete or sound.

**Conceptual limitations:** Parsed preprocessor macros follow the pattern `#define identifier(parameters) macrocontent`, where the parameters are optional. Based on our concept, we can easily identify calls of function-like macros, but macros that just replace characters are not detected. For example, if a macro PI is defined that replaces all appearances of PI with 3. 14, there will be no edges from all PI occurrences to the macro. It could be possible to conduct an analysis to find these edges, but this is costly. Therefore, we decided not to implement such an analysis. Instead, a developer could use a custom query for this purpose.

**Limitation of the current status of the implementation:** The tool implementation is still ongoing. Currently, most, but not all, node types have a specific rule set. Further, the variability analysis outside of functions is limited, as there was no reusable data- and control-flow analysis. However, these are no fundamental limitations of our approach.

## 4 RELATED WORK

Typically, program slicing is done based on program or system dependence graphs [18]. These graphs are computed through static analysis of the source code. Therefore, we structure the related work in two parts: In the first part, we highlight other approaches that can compute a code representation like an AST, control-flow or data-flow graph for preprocessor-annotated product lines without resolving the variability. In the second part, we discuss slicing approaches that aim at slicing software product lines.

**Code representation:** The tool SuperC [9] generates a variability-aware AST from preprocessor-annotated code. It resolves preprocessor annotations, except from conditionals, during the lexing process and enriches the AST with included file contents and expanded macros. The `#ifdef`-statements are represented by static

choice nodes inside the AST. The choice nodes contain two child-nodes, one for the true condition and one for the false condition. While this code representation allows for some variability-aware static analyses, it is not suitable for variability-aware semantic slicing, as it does not fulfill **(C2)**. First, we need all the preprocessor statements themselves as a basis for setting them either as entry point or for retrieving them in the resulting slice. Second, the AST representation alone is not sufficient for program slicing as we also need control- and data-flow information.

The tool TypeChef [13] computes a variational AST, like SuperC does. The main difference is that TypeChef also resolves the `#ifdef`-statements. This results in an AST with presence conditions instead of the statical choice nodes from SuperC. There are some newer extensions that allow to conduct control- and data-flow analysis [1, 15]. While TypeChef's code representation allows the application of program slicing, it is not suitable for variability-aware semantic slicing, as it lacks the same information as SuperC: There are no preprocessor statements in the resulting graphs and therefore **(C2)** is not fulfilled.

Brabrand et al. [6] present an approach that conducts feature-sensitive intraprocedural data-flow analysis. They annotate the data-flow graph with configuration sets that determine under which conditions a certain node is reached. The configuration sets are derived from the preprocessor statements, whereas the nodes contain, for example, the C code. This approach is, like SuperC and TypeChef, not applicable as basis for variability-aware semantic slicing, as it contains again no preprocessor statements in the resulting graph and, thus, does not fulfill **(C2)**. Furthermore, an additional control-flow analysis is needed.

The tool SPL<sup>LIFT</sup> [5] allows to automatically adapt static analyses formulated with the IFDS framework [16] to software product lines. It is applicable to product lines implemented with the Colored Integrated Development Environment (CIDE) [12]. To make their approach applicable to C product lines, the authors state that they plan to use TypeChef. Therefore, this approach will not fulfill **(C2)**.

srcML is a tool that parses code, including preprocessor statements, into an XML representation [7]. This representation can then be used to build, for example, an AST, control- or data-flow graph, but these three graphs are all not part of the tool itself. Program slicing requires data- and control-flow analysis, thus, the srcML code representation alone is not sufficient for program slicing.

Joern [22] is a tool for vulnerability-analysis of software and operates based on code property graphs. This graph combines an AST with control- and data-flow information, making it a proper base for program slicing. However, Joern's code representation completely ignores preprocessor statements. While this allows the tool in principle to analyze product lines implemented in C and C++, it makes its code representation insufficient for variability-aware program slicing.

**Slicing approaches:** The slicing approach by Kanning et al. [11] is applicable to software with preprocessor variability. Their approach is built upon TypeChef. As a result and in contrast to our approach, it can not slice with preprocessor statements as entry point or output a slice that contains preprocessor statements. Therefore, it does not fulfill **(C1)**.

The tool Emergo [17] reuses the work of Brabrand et al. [6] and Bodden et al. [5] to realize emergent feature interfaces. Based on a maintenance point, which is comparable to our entry point, the

tool shows the developer the relevant lines of code. While this approach measurably helps developers to understand the impact of their changes (forward-slicing), it does not provide backward-slicing. Furthermore, the approach inherits the limitations of the reused approaches and does therefore not fulfill **(C1)**.

srcSlice [2] is a lightweight slicing tool that builds upon srcML. However, it is only used for forward slicing and while it is very fast, it lacks accuracy, as it does not perform a complete computation of the dependence graphs.

Angerer et al. present an approach for configuration-aware change impact analysis [3]. Their approach is built upon a conditional system dependence graph. In contrast to our approach, they target load-time variability and rely on configuration files. Furthermore, their change impact analysis is bound to only execute forward slicing.

The semantic slicing approach realized with the tool CSlicer [14] has comparable goals to our approach, but it is based on a completely different test-based approach. This makes it heavily dependent on the quality of the test cases. Further, it is only available for Java programs and slices commits, not the whole program.

In summary, there exist a number of comparable approaches for variability-aware code representations and slicing techniques. The approaches most similar to our approach are the tools srcML and Joern. While the code representation of srcML is the only one that contains preprocessor statements, the dependence analyses of Joern are far more complete. Furthermore, Joern's code property graph was comparatively easy to enhance with preprocessor statements. Thus, we decided to create our approach as an extension of the Joern implementation.

## 5 CONCLUSION

The semantic analysis of product lines is a complex activity for developers, but an absolute necessity in the context of software evolution. Especially with preprocessor-based product line implementations, the human understanding of the product line code is very cumbersome. While program slicing is a well-known approach for supporting program comprehension, so far there has been no adequate support for variability-aware program slicing of preprocessor-based code. With the work that we introduce here, we present an initial approach to address this gap.

Our approach introduces the novel concept of *variability-aware code property graphs*. This extends earlier work on code property graphs that combine ASTs, data-flow, and control-flow graphs into a single data structure [22]. We illustrated the significant capabilities of this concept with an example. The approach is available as initial tooling<sup>1</sup> that is highly adaptive, as the inference rules for slicing are fully customizable. Hence, it can be more regarded as an open infrastructure for the automated variability-aware slicing of code than a single tool.

As future work, we plan to finish the current tool implementation, which is functional, but not complete. In particular, the rules for all node types of the AST and the intraprocedural analyses are not fully implemented. We also plan to evaluate the capabilities of our approach and the accompanying tool in the context of existing C-based product line case studies.

<sup>1</sup><https://github.com/LPhD/Jess>

## REFERENCES

- [1] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 4 (Nov. 2018), 18:1–18:33. ACM, New York, NY, USA.
- [2] Hakam W. Alomari, Michael L. Collard, Jonathan I. Maletic, Nouh Alhindawi, and Omar Meqdadi. 2014. srcSlice: Very Efficient and Scalable Forward Static Slicing. *Journal of Software: Evolution and Process* 26, 11 (Nov. 2014), 931–961. <https://doi.org/10.1002/sm.1651>
- [3] Florian Angerer, Andreas Grimmer, Herbert Prähofer, and Paul Grünbacher. 2019. Change impact analysis for maintenance and evolution of variable software systems. *Automated Software Engineering* 26 (Feb. 2019), 1–45. <https://doi.org/10.1007/s10515-019-00253-7>
- [4] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, Baltimore, MD, USA, 257–269.
- [5] Eric Bodden, Tarsis Toledo, Marcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPLIFT – Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 355–364.
- [6] Claus Brabrand, Márcio Ribeiro, Társis Tolédo, and Paulo Borba. 2012. Intraprocedural dataflow analysis for software product lines. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development - AOSD '12*. ACM Press, Potsdam, Germany, 13. <https://doi.org/10.1145/2162049.2162052>
- [7] Michael L. Collard, Michael J. Decker, and Jonathan I. Maletic. 2011. Lightweight Transformation and Fact Extraction with the srcML Toolkit. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. IEEE, Williamsburg, VA, USA, 173–184. <https://doi.org/10.1109/SCAM.2011.19>
- [8] Duc Le, E. Walkingshaw, and M. Erwig. 2011. #ifdef confirmed harmful: Promoting understandable software variation. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Pittsburgh, PA, 143–150. <https://doi.org/10.1109/VLHCC.2011.6070391>
- [9] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 323–334.
- [10] Lea Gerling. 2018. Automated migration support for software product line co-evolution. In *Proceedings of the 40th International Conference on Software Engineering Companion Proceedings - ICSE '18*. ACM Press, Gothenburg, Sweden, 456–457. <https://doi.org/10.1145/3183440.3183441>
- [11] Frederik Kanning and Sandro Schulze. 2014. Program Slicing in the Presence of Preprocessor Variability. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, Victoria, BC, Canada, 501–505. <https://doi.org/10.1109/ICSME.2014.82>
- [12] Christian Kästner. 2012. Virtual Separation of Concerns: Toward Preprocessors 2.0. *it - Information Technology* 54, 1 (Feb. 2012), 42–46. <https://doi.org/10.1524/itit.2012.0662>
- [13] Christian Kästner, Paolo G Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, Portland, Oregon, USA, 805–824.
- [14] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2017. Semantic slicing of software version histories. *IEEE Transactions on Software Engineering* 2, 44 (2017), 182–201.
- [15] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. ACM Press, Saint Petersburg, Russia, 81. <https://doi.org/10.1145/2491411.2491437>
- [16] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*. ACM Press, San Francisco, California, United States, 49–61. <https://doi.org/10.1145/199448.199462>
- [17] Márcio Ribeiro, Paulo Borba, and Christian Kästner. 2014. Feature maintenance with emergent interfaces. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM Press, Hyderabad, India, 989–1000. <https://doi.org/10.1145/2568225.2568289>
- [18] Josep Silva. 2012. A vocabulary of program slicing-based techniques. *Comput. Surveys* 44, 3 (June 2012), 1–41. <https://doi.org/10.1145/2187671.2187674>
- [19] Henry Spencer and Geoff Collyer. 1992. #ifdef Considered Harmful, or Portability Experience With C News. In *Proceedings of the USENIX Summer 1992 Technical Conference*. USENIX Association, San Antonio, Texas, 185–197.
- [20] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (June 2014), 1–45. <https://doi.org/10.1145/2580950>
- [21] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, 439–449.
- [22] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, 590–604. <https://doi.org/10.1109/SP.2014.44>

## A ARTIFACT DESCRIPTION

This section describes three artifacts and how to use them: The **example code** from Listing 1, the variability-aware semantic slicing prototype **Jess** and the semantic slicing **results**. The example code and a visualization of the results are both contained in the same location as the tool Jess: <https://github.com/LPhD/Jess>. The next section describes the installation of Jess and Section A.2 describes the usage of Jess and how the results from the paper can be reproduced.

### A.1 Installation

Currently, Jess is only validated for Linux environments. The results in the paper were generated using Ubuntu Desktop Version 16.04.5 and release v0.2 of Jess. We can not guarantee the reproducibility of the results for other versions of Jess or Linux. Before Jess can be installed, some external dependencies must be installed first. A detailed explanation of the dependencies, the installation process and some technical background can be found in Jess' official documentation: <https://jess.readthedocs.io/en/stable/>. To install Jess, clone the repository and run the installation script:

```
sudo apt-get update
sudo apt-get install openjdk-8-jdk gradle python3
sudo apt-get install python3-setuptools python3-dev
sudo apt-get install graphviz graphviz-dev git
git clone --branch v0.2
https://github.com/LPhD/Jess.git
cd Jess
./build.sh
```

After the installation is finished, add the local user directory to the path. This allows to use the Jess scripts from every location.

```
export PATH="$PATH: /.local/bin"
```

### A.2 How to use

Jess generates a variability-aware code property graph, as described in Section 3.1. For this purpose, Jess parses a given software project into a graph database based on TitanDB<sup>2</sup>. This database can then be queried using the Gremlin<sup>3</sup> query language. To generate and query a code property graph, the Jess server must be running.

```
cd path/to/Jess
./jess-server.sh
```

In a second terminal, a project can be imported. Here we demonstrate the import of the example project described in Listing 1. Therefore, the source code of the project must be first packed as a tarball. Then jess-import is invoked to import the project. This will upload the tarball to the server, unpack it, parse the code and create a new project named SPLC and its corresponding graph database.

```
cd path/to/Jess
```

```
tar -cvzf SPLC testProjects/SPLC/
```

```
jess-import SPLC
```

After the import is finished, the semantic slicing process can be invoked on the SPLC project to generate the results described in Figure 3. The variability-aware semantic slicing approach is implemented as a Python 3 script named SUI.py. The script can be configured for the user's needs. More details on the configuration are described in the official documentation. For the replication of the results described in the paper, the current configuration is sufficient.

```
cd customScripts/
```

```
python3 SUI.py
```

The results are stored into the SemanticUnit folder as a SemanticUnit.dot file that contains textual descriptions for each node and edge, and as SemanticUnit.png file that contains a graphical representation of the results. Note that the generated files contain all AST children in contrast to the version in Figure 3, where most of the details are left out to simplify the visualization. However, the top level nodes that directly appear in the source code are the same.

<sup>2</sup><http://titan.thinkaurelius.com/>

<sup>3</sup><https://tinkerpop.apache.org/docs/3.0.1-incubating/>

# Applying Product Line Engineering Concepts to Deep Neural Networks

Javad Ghofrani

Faculty of Informatics/Mathematics  
HTW University of Applied Sciences  
Dresden, Germany  
javad.ghofrani@gmail.com

Anna Lena Fehlhaber

Leibniz University Hanover  
Hanover, Germany  
fehlhaberlena@gmail.com

Ehsan Kozegar

Faculty of Engineering (Eastern Guilan)  
University of Guilan  
Guilan, Iran  
kozegar@guilan.ac.ir

Mohammad Divband Soorati

University of Lübeck  
Lübeck, Germany  
divband@iti.uni-luebeck.de

## ABSTRACT

Deep Neural Networks (DNNs) are increasingly being used as a machine learning solution thanks to the complexity of their architecture and hyperparameters—weights. A drawback is the excessive demand for massive computational power during the training process. Not only as a whole but parts of neural networks can also be in charge of certain functionalities. We present a novel challenge in an intersection between machine learning and variability management communities to reuse modules of DNNs without further training. Let us assume that we are given a DNN for image processing that recognizes cats and dogs. By extracting a part of the network, without additional training a new DNN should be divisible with the functionality of recognizing only cats. Existing research in variability management can offer a foundation for a product line of DNNs composing the reusable functionalities. An ideal solution can be evaluated based on its speed, granularity of determined functionalities, and the support for adding variability to the network. The challenge is decomposed in three subchallenges: feature extraction, feature abstraction, and the implementation of a product line of DNNs.

## CCS CONCEPTS

- Computing methodologies → Neural networks; Artificial intelligence;
- Software and its engineering → Software product lines.

## KEYWORDS

Software product lines, variability, deep neural networks, transfer learning, machine learning

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336321>

## ACM Reference Format:

Javad Ghofrani, Ehsan Kozegar, Anna Lena Fehlhaber, and Mohammad Divband Soorati. 2019. Applying Product Line Engineering Concepts to Deep Neural Networks. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3336294.3336321>

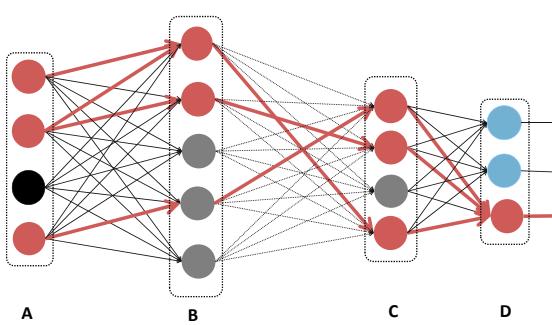
## 1 INTRODUCTION

Software product lines (SPLs) [16] or software families [23] are software intensive systems which are developed using a managed set of reusable assets<sup>1</sup>. Software product line engineering methods enable software engineers to generate different software products while supporting the variety and commonalities in communication and behavior of software assets. Using customization in behavior and constraints on communications of software assets facilitate the organizations to develop their products faster with supporting a larger spectrum of requirements of their customers. Advances in reusing software assets have shown that SPLs optimize implementation while saving time and costs. These factors make the significant difference in the field of production and software industry. A software asset is a well-designed software system or subsystem that is engineered or refined to get easily integrated into the development of a SPL while providing a functionality within the software products. Software assets can have new behaviors and are able to communicate with other components. Feature-oriented software development [1] is one of the most efficient ways to realize SPLs in which the assets are described as features providing functionality to the system. Adding and removing features can affect the entire structure of the software product. Feature models are used to describe a SPL including the features and their relation to each other, i.e., constraints [4, 11].

Deep neural network (DNN) [18] is a buzz word in different fields of computer science which attracted the interest of many experts, e.g., image classification, activity recognition, object detection, automatic speech recognition, machine translation, etc. Neural networks cover a subset of the machine learning methods to perform classification and prediction tasks, based on the input values in a computational system. The advantage of neural networks compared to the algorithms are their relative tolerance to the noise in their input values. Furthermore, neural networks can

---

<sup>1</sup><https://www.sei.cmu.edu/>



**Figure 1: Example of DNN in which some parts are activated with a certain input value. Input (A), hidden (B&C), and output layers (D) are shown. The activated part of the neural network is highlighted in red.**

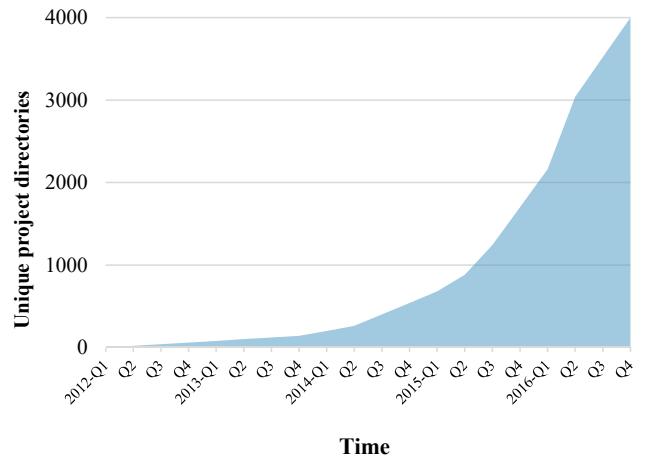
be trained only with the data and do not require the development of an algorithm or mathematical solutions, which is the base of many functionalities in software systems. Recently, DNNs have grown dramatically thanks to cloud computing and analytic methods in big data. In DNNs, the process of finding any correlation between input and output will be performed in different levels of abstraction. Due to the necessity of finding structures and patterns in the world, the machine is forced to transform observations into data. Therefore, it is possible to identify different and more complex patterns. Various types of DNNs are developed based on their usage and input types. The commonly used type of DNNs are convolutional networks<sup>2</sup> [18] which are used in image processing tasks (e.g., NVIDIA's autonomous driving project<sup>3</sup>).

A DNN consists of an architecture (network layers) and its hyperparameters (weights of the edges). The basic unit of the architecture of a neural network is called “neuron”. A set of neurons form a layer in the architecture of neural networks. A neural network consists of one or many layers with a matrix of neurons. “Edges” are the other basic units within the architecture of DNNs that connect the neurons in the neighboring layers. Figure 1 illustrates the structure of a neural network. DNNs receive the inputs from the first layer (input layer) and return the results via the last layer (output layer) of their architecture. A matrix of float values will be fed to the input layer to activate the network. The network maps the input values to a matrix in the output layer. In an ideal case, a node of output with maximum value reflects the decision of the network based on the input values. This mechanism will be used for prediction or classification tasks. Training a neural network initializes the neurons and sets the weights on edges. The weight values assigned to the edges are called hyperparameters. Using Keras library<sup>4</sup> one can define the architecture of a network in python and save or load the hyperparameters of neural network using a separate file with *h5* extension.

<sup>2</sup><http://cs231n.github.io/convolutional-networks/>

<sup>3</sup><https://www.nvidia.com/en-us/self-driving-cars/>

<sup>4</sup><https://keras.io/>



**Figure 2: Growing use of deep neural networks at Google [5]**

*Motivation:* DNNs have shown that they can be applied to almost any domain if a set of training data is available. Instead of writing new programs, the data will be collected, and the neural network will be trained. Image processing, text summarization, text translation, and IoT (such as autonomous driving and smart homes) are making DNNs an alternative or even competitor to the software systems.

There is a trend for using DNNs in various fields such as natural language processing, image understanding, even android apps (See Figure 2). There is a potential for deep neural networks to act as a replacement for software systems in various fields. However, for each new task, the neural networks should be rebuilt, re-trained, justified or re-tuned.

*Contribution:* Transferring the knowledge of software engineering to the new emerging concept of DNNs is an opportunity that we try to provide. We aim for connecting the communities of machine learning and variability management to motivate both fields in performing interdisciplinary research for applying reuse in DNNs which leads to save of time, resources and cost of re-training DNNs for new tasks. From a software engineering perspective, we are interested in factors such as cost (here for providing data and computational power for training DNNs), time-to-market, and quality. For example, eliminating ineffective parts of the neural network (if possible and detected) reduces the memory and power consumption and makes the system more efficient. This paper is a way to investigate the feasibility and performance of these factors.

The remainder of this paper is structured as following. Section 2 explains the details of the challenge and evaluation methods for solutions. Section 3 introduces some related work, and Section 4 discusses the benefits of applying variability management methods to DNNs. Finally, we summarize our paper in Section 5.

## 2 THE CHALLENGE

In this paper, we ask the community of SPL researchers to apply concepts of SPLs to DNNs with well-defined commonalities and variabilities. By using the term “feature” we refer to the concept of feature-oriented software product lines [1]. In this domain, the

features are the functionalities of software systems. We consider the functionalities of DNNs in analogy to features in software systems. As mentioned before, one of the use cases of the neural networks is in image processing as classifiers. Let us assume a neural network that identifies and classifies animals such as dogs and cats. In this example, the input layer of DNN receives a matrix of a 2-D image. In the elementary layers, the primary components including the edges and the corners are extracted. Two neurons in the output layer indicate the probabilities of having a cat or a dog in the input image.

A functionality of a DNN can be the prediction of the class of a certain object in an image that is given as the input. A DNN capable of recognizing (predicting or classifying) three types of animals in a picture has then three functionalities. We explain our notion of functionalities with an example of VGG 16 DNN. This network is pre-trained on the ImageNet dataset capable of recognizing 1000 different object categories (e.g., tiger cat, blue tick, sea lion, sorrel, ...). For instance, the capability to recognize a Persian cat and a zebra are two different functionalities. A list of all 1000 categories can be found online<sup>5</sup>.

The term “feature” appears in both software engineering and for feature mapping in deep learning domains. We avoid the confusion by using features only in the context of software engineering and the term functionality when we discuss the neural networks. Both terms—feature and functionality—refer to the same concept but in different domains.

The challenge is in three levels of increasing complexities. The participants of the challenge should report their solutions, results and faced challenges during the adoption process. We welcome suggestions from the participants regarding software configuration methods that allow DNNs to be configured for reusability without additional re-training.

## 2.1 First Challenge: Feature Extraction

Since the extraction and specification of features is a challenge in product line engineering, the participants are asked to propose solutions for extraction of functionalities from DNNs. We use the term “parts” in this paper as subsets of a neural network’s architecture [7]. We also consider a subset of the hyperparameters—weights of the edges—as parts. Figure 3 highlights a part in a neural network. We explain this challenge with an example of a simple pre-trained network capable of recognizing cats and dogs in an input image. The part in charge of recognizing a cat consists of the subsets of the network’s architecture and hyperparameters that are activated if a given image to the network contains a picture of a cat.

*Challenge:* We ask participants to take a sample pre-trained network which can be found online in imageNet<sup>6</sup> with 1000 classification functionalities. Examples of such models are available in Keras documentation<sup>7</sup> [2]. Some of the pre-trained models are Xception [3], VGG16 and VGG19 [20], ResNet50 [8], InceptionV3 [21], and MobileNet [9].

We see two possibilities for solving this challenge; first, trying to identify the common parts between the members of a set

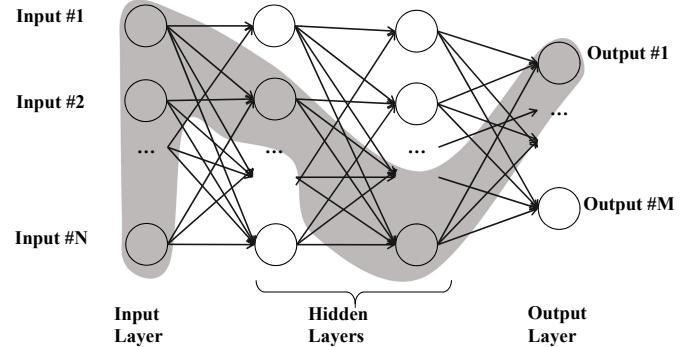


Figure 3: An example of a part of a DNN as a subset of its architecture

of trained neural networks which are activated if a certain input (e.g., *persian\_cat*) is given to these networks. For example, taking ResNet50, VGG19 and VGG16 as a set and trying to feed all of them with the same group of input images and identifying the parts of network which will be activated or used to take a decision. The commonalities between the identified parts of the networks have to be found next. The second possibility is to take one network to specify the boundaries of functionalities within the network. For example, we can take VGG19 and try to determine responsible parts for recognition of each category of objects, e.g., responsible parts for recognizing Persian cat.

Solutions could be manual, semi-automated or fully automated to extract the corresponding parts which are responsible for each functionality.

*Evaluation:* Solutions for the raised challenge can be evaluated based on several criteria. We favor the solutions with least time taken during the extraction. Another metric to minimize is the level of human interventions. The solution with the least time consumption and human intervention is considered the highest quality solution.

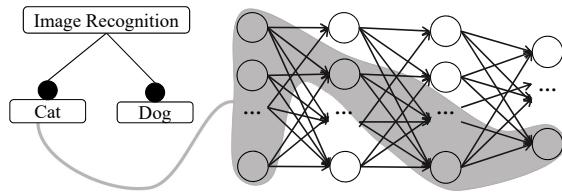
## 2.2 Second Challenge: Feature Abstraction

The principle of uniformity [1] denotes that, not only the code fragments are considered as reusable assets in SPLs but the other artifacts such as documentation—including models and specifications—and test cases are reusable assets as well. Feature models [10] and orthogonal variability model [16] provide a level of abstraction to hide the complexity of the implementation of the features. These models handle the dependencies between features of product lines and the reusable assets required to implement them (including every required artifacts). In our notion of reusable assets of a DNN, each subset of a DNN that is responsible for a certain functionality should be bound to a feature in a feature model. This enables the application engineers to include or exclude some of the functionalities of a DNN to create a new DNN with reduced number of functionalities. Figure 4 shows an example of features of a product line bound to their implementations which are the parts of a DNN.

<sup>5</sup><http://image-net.org/challenges/LSVRC/2014/browse-synsets>

<sup>6</sup><http://www.image-net.org/>

<sup>7</sup><https://keras.io/applications/>



**Figure 4: Example of binding the parts of a DNN to a feature model**

In this example, the corresponding part for cat recognition in the input image is bound to the cat feature in feature diagram.

Various mechanisms are developed to facilitate the binding of feature in feature models to their corresponding reusable assets with less implementation efforts. A set of well-established methods are presented in feature oriented software product lines by Apel et al. [1]. These methods support the implementation of denoted variabilities and commonalities by a feature model. However, handling the parts of a DNN as reusable assets is not considered in existing methods.

*Challenge:* In this level of the challenge, we are expecting the participants to introduce a method to assign the extracted parts of DNN to a feature model. Selecting or deselecting a feature in feature model should include or exclude the corresponding part of the DNN that are responsible for mentioned functionality by that feature. For this purpose, an implementation of a feature model using existing feature modeling tools (e.g., FeatureIDE [12]) is required.

*Evaluation:* We expect the participants to apply known methods for implementing commonalities and variabilities from SPLs to DNN. Apel et al. introduced and explained these methods in their book on “Feature-Oriented Software Product Lines” [1]. These methods are Parameters, Design Patterns, Frameworks, Components and Services, Version-Control Systems, Build Systems, Preprocessors, Feature-Oriented Programming, Aspect-Oriented Programming, Delta-Oriented Programming, and Context-Oriented Programming. Referring to each of these methods to implement the variability in DNN will be counted as a positive point for evaluation. Furthermore, introducing a new method for implementing the variability in DNNs is considered as a plus as well. For example, a possible solution, that is implemented with Parameters method, takes a pre-trained neural network (e.g. VGG16) and a list of parameters to enable or disable the functionalities of that network (e.g., Cat=true, Dog=false, ...), and generates a new network with desired functionalities. This type of solution takes a positive point in the evaluation. Reengineering a network to apply a design pattern for adding or removing some observer for certain functionalities is also a plus. Granularity of the extracted parts is the next metric for efficiency of a proposed solution. The number of the functionalities included in an extracted part of the network defines the granularity. We prefer the solutions with higher number of parts that explicitly perform a set of few functionalities. Assume  $A$  is a solution that extracts two parts from the given network, the first part performs one

functionality and the second part performs two (i.e.,  $P_A = \{1, 2\}$ ). Similarly, there are two other solutions ( $B$  and  $C$ ) with  $P_B = \{1, 1, 1\}$  and  $P_C = \{1, 2, 1\}$ . Based on our evaluation method,  $C$  is the best solution followed by  $B$ . Equation 1 is a formal definition for our quantitative approach to evaluate the solutions.

$$\text{Score} = \sum_{x \in P} x \times \sum_{x \in P} \frac{1}{x} \quad (1)$$

where  $P$  is the set of functionalities performed by the parts in a solution. The scores of the solutions in our previous example will be  $\text{Score}(A) = (1 + 2) \times (1/1 + 1/2) = 4.5$ ,  $\text{Score}(B) = 3 \times 3 = 9$ , and  $\text{Score}(C) = 4 \times 2.5 = 10$ .

### 2.3 Third Challenge: Product Line of DNNs

This challenge corresponds to the promising capability of SPLs to generate software products from existing well managed software assets.

*Challenge:* At this stage of the challenge, participants are asked to create a product line from DNNs. To solve this challenge, the results of the previous steps should be put together in order to produce a composed neural network. The functionalities of the generated DNN is the same as the functionalities of the neural networks extracted, abstracted, and decomposed in the previous stages of the challenge. Suppose  $M$  is a DNN that classifies images of dogs and cats and  $N$  is another DNN that classifies crows and eagles. There has to be the possibility of generating a network that supports a combination of these functionalities. From  $M$  and  $N$  a new network  $O$  can be built that identifies crows and dogs or another network  $P$  that recognizes crows, eagles, and cats but dogs.

*Evaluation:* The size of the generated neural network should be minimal. Any redundancy increases the size of the generated neural network. The accuracy of the final neural networks should not be much lower than the base neural networks for the same input. The solutions should have a feature model to bind the variable and common assets or parts of the network.

## 3 RELATED WORK

Kruger et al. [13] proposes the challenge of generating a product line from APO-Game clones. They support extracting features in a set of cloned software products which is different to the extraction of features in the application domain of machine learning techniques. The state-of-the-art lacks the opportunities emerging from deep learning for the field of SPLs.

Deep transfer learning [15] is a method to reinforce or induce a trained DNN to reuse it for a different task [22]. DNNs trained with a general dataset for a general task are refined or re-tuned for a special task. For example, a DNN trained for a general image processing task is then refined for face recognition (as a subset of image processing domain). This method reduces the time and costs for training DNNs from scratch for each new task. Companies like Google train the models and provide them so that everyone can reuse the pre-trained neural networks in their own domains such as image processing<sup>8</sup> or natural language processing<sup>9</sup>. Transferred

<sup>8</sup><https://image-net.org>

<sup>9</sup><https://wordnet.princeton.edu>

learning is an effective approach if the available dataset for training the network is not big enough or the infrastructure does not provide the required computational power to perform the training. In transferred learning a neural network can be reused as a subset of another network only after being re-training with the most recent data. In our challenge, we want to apply the reuse possibilities for neural networks without the need for re-training. Catastrophic forgetting in the neural networks is a known issue [14, 17] where the networks forget the old skills while training for new skills due to the change in the weights of the edges of the networks. Ellefsen et al. [6] apply the modular neural networks [7] to solve the issue by reinforcing them with various solutions. The possibility to apply these methods for the DNNs is not investigated yet.

## 4 DISCUSSION

Sculley et al. [19] investigated the drawbacks of using machine learning, specifically the DNNs, as a part of a software system. They address the risk factors raised by any change in the requirements or specifications of system that is based on machine learning techniques. Knowing this issue, here we explain how our notion of applying SPL concepts to DNNs help to overcome this problem. Separating and abstracting the functionalities of a DNN allow a new network to form from a combination, composition, or subtraction of the network functionalities without re-training. In case of an erosion of boundaries, only the affected parts need to be replaced. It takes less time, cost, and effort to train and test the new neural network. The same applies to the problem of Entanglement. Despite of CACE (change anything, change everything) concept, any accuracy improvement in certain parts of the network can be performed without affecting or re-training the whole network. Correction cascades can be prevented by our notion of features.

Instead of using the old models as a basis for new ones, the parts of the system that are not relevant anymore will be replaced. Dependencies to the old network can be prevented and the undeclared consumers can be covered partly with our method. When a functionality changes, the undeclared consumers are affected as well, if they use the altered functionalities. Hidden feedback loops and data dependencies will be easier to detect and remove if the corresponding parts of the network for each feature or functionality is already determined.

Changes in the external world is a general problem of the software systems. In DNNs, any change in the requirements of the system leads to re-training the whole system even when a change is in one or some functionalities of the network. This problem could be handled if the relation between network parts and their functionality is clearly explained. Under these circumstances, only the parts of the network will be changed or updated that their corresponding functionalities are in correlation with changes in outside world. The configuration issues and variety of system-level anti-patterns mentioned by Sculley et al. [19] are not supported directly by our challenge proposal. One can implement a DNN for each functionality, but it would suffer from the similar shortages in the methods for service composition [1] that leads to a redundancy and waste of memory and energy.

## 5 SUMMARY

In this paper, we proposed the challenge of applying techniques and concepts of SPLs to the DNNs. We challenge the researchers in both fields to find a solution to specify, extract, and combine the functionalities from the DNNs to enable the reuse in building a product line. Solving this challenge can improve the preparation of data, time, and costs of the resources for training a DNN for every given task. The proposed solutions will be evaluated based on the promised benefits of the product line engineering—time, cost, and quality. The solutions will be collected in a repository to provide a benchmark for evolution and evaluation of the new methods of applying concepts of the SPLs to the DNNs.

## REFERENCES

- [1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-oriented software product lines*. Springer.
- [2] François Chollet. 2017. Keras Documentation. 2016.
- [3] François Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. *arXiv preprint* (2017), 1610–02357.
- [4] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. 2012. Cool features and tough decisions: a comparison of variability modeling approaches. In *Proceedings of the sixth international workshop on variability modeling of software-intensive systems*. ACM, 173–182.
- [5] Jeff Dean. 2017. *AIFrontiers:Trends and Developments in Deep Learning Research*. <https://www.slideshare.net/AIFrontiers/jeff-dean-trends-and-developments-in-deep-learning-research>
- [6] Kai Olav Ellefsen, Jean-Baptiste Mouret, and Jeff Clune. 2015. Neural modularity helps organisms evolve to learn new skills without forgetting old skills. *PLoS computational biology* 11, 4 (2015), e1004128.
- [7] Frédéric Gruau. 1994. Automatic definition of modular neural networks. *Adaptive behavior* 3, 2 (1994), 151–183.
- [8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.
- [9] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).
- [10] Kyu C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.
- [11] Kyu C Kang, Jaejoon Lee, and Patrick Donohoe. 2002. Feature-oriented product line engineering. *IEEE software* 19, 4 (2002), 58–65.
- [12] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A tool framework for feature-oriented software development. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 611–614.
- [13] Jacob Krüger, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. 2018. Apo-games: a case study for reverse engineering variability from cloned Java variants. In *Proceedings of the 22nd International Conference on Systems and Software Product Line-Volume 1*. ACM, 251–256.
- [14] Michael McCloskey and Neal J Cohen. 1989. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, Vol. 24. Elsevier, 109–165.
- [15] Sinno Jialin Pan, Qiang Yang, et al. 2010. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2010), 1345–1359.
- [16] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.
- [17] Roger Ratcliff. 1990. Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. *Psychological review* 97, 2 (1990), 285.
- [18] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural networks* 61 (2015), 85–117.
- [19] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. In *Advances in neural information processing systems*, 2503–2511.
- [20] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
- [21] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In

- Proceedings of the IEEE conference on computer vision and pattern recognition.* 2818–2826.
- [22] Lisa Torrey and Jude Shavlik. 2010. Transfer learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*. IGI Global, 242–264.
- [23] David M Weiss and Chi Tau Robert Lai. 1999. *Software product-line engineering: a family-based software development process*. Vol. 12. Addison-Wesley Reading.

# Product Sampling for Product Lines: The Scalability Challenge

Tobias Pett  
t.pett@tu-braunschweig.de  
TU Braunschweig  
Germany

Sebastian Krieter  
sebastian.krieter@ovgu.de  
University of Magdeburg  
Germany

Thomas Thüm  
t.thuem@tu-braunschweig.de  
TU Braunschweig  
Germany

Malte Lochau  
malte.lochau@es.tu-darmstadt.de  
TU Darmstadt  
Germany

Tobias Runge  
tobias.runge@tu-braunschweig.de  
TU Braunschweig  
Germany

Ina Schaefer  
i.schaefer@tu-braunschweig.de  
TU Braunschweig  
Germany

## ABSTRACT

Quality assurance for product lines is often infeasible for each product separately. Instead, only a subset of all products (i.e., a sample) is considered during testing such that at least the coverage of certain feature interactions is guaranteed. While pair-wise interaction sampling only covers all interactions between two features, its generalization to  $t$ -wise interaction sampling ensures coverage for all interactions among  $t$  features. However, sampling large product lines poses a challenge, as today's algorithms tend to run out of memory, do not terminate, or produce samples, which are too large to be tested. To initiate a community effort, we provide a set of large real-world feature models with up-to 19 thousand features, which are supposed to be sampled. The performance of sampling approaches is evaluated based on the CPU time and memory consumed to retrieve a sample, the sample size for a given coverage (i.e. the value of  $t$ ) and whether the sample achieves full  $t$ -wise coverage. A well-performing sampling algorithm achieves full  $t$ -wise coverage, while minimizing the other properties as best as possible.

## CCS CONCEPTS

- Software and its engineering → Software product lines.

## KEYWORDS

software product lines, product line testing, product sampling, real-world feature models

## ACM Reference Format:

Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3336294.3336322>

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-7138-4/19/09...\$15.00  
<https://doi.org/10.1145/3336294.3336322>

## 1 INTRODUCTION

Modern software engineering struggles with increasing requirements regarding finer customization and faster time to market. A common solution is to design systems as software product lines. This way a collection of customized products can be build based on a common core [3]. However, the variability contained in a software product line poses a potentially large number of possible products. Analyzing all products individually is infeasible in most cases [30]. Therefore, quality assurance is often performed on a small subset of product configurations, which presumably covers a sufficient amount of functionality of the product line.

A common technique to build a subset of products is *combinatorial interaction sampling (CIT)* [17]. The goal of combinatorial interaction testing is to build samples, which achieve  $t$ -wise interaction coverage between sets of features.  $T$ -wise interaction coverage requires that every possible combination of  $t$  features is covered by at least one product in the sample. Commonly used coverage criteria include feature-wise coverage ( $t = 1$ ), pair-wise coverage ( $t = 2$ ), or three-wise coverage ( $t = 3$ ).

Over the years, many sampling algorithms have been developed to generate samples that achieve feature interaction coverage. To generate a sample different input artifacts such as a feature model, implementation artifacts [11, 23, 25] or expert knowledge [7, 9] can be used [30]. However, the majority of sampling algorithms uses a feature model as single input artifact [30]. Hence, our challenge focuses on sampling algorithms using a feature as single input for sampling. Sampling algorithms drastically reduce the number of products to be tested, compared to testing all possible products individually. However, when applied to large product lines from real-world applications with hundreds or even thousands of features, they often run out of memory, do not terminate, produce unnecessary large samples to test, or the calculation takes to much time [4, 8]. Facing those scalability issues is a recent challenge of product line research [15].

To assure the quality of real-world product lines is already a challenging task by considering only one version. However, a product line evolves throughout its whole life cycle, based on changing environments or newly required functionality [19]. To assure that no unwanted behavior was introduced by the changes, a new sample needs to be generated for each product line version. Hence, the sampling scalability challenge needs to be handled again for each product line version. Until now, no sampling algorithm considers the evolution of product lines or previously calculated samples to

speed up the sampling process for retesting product-line functionality [30].

We contribute to the research of product-line sampling by providing a collection of feature model evolution histories of real-world product lines. Our collection includes a selection of feature models for the history of the Linux kernel<sup>1</sup>, including 400 versions<sup>2</sup> with over 19,000 features. Furthermore, we provide the evolution history of Automotive02<sup>3</sup> (four product-line versions with about 14,000 to 19,000 features), and FinancialServices01<sup>4</sup> (ten product-line versions with about 500 to 700 features). We argue that our provided product lines are well suited as subjects for the challenge of scalability of t-wise sampling algorithms, based on their size and use in different industrial environments.

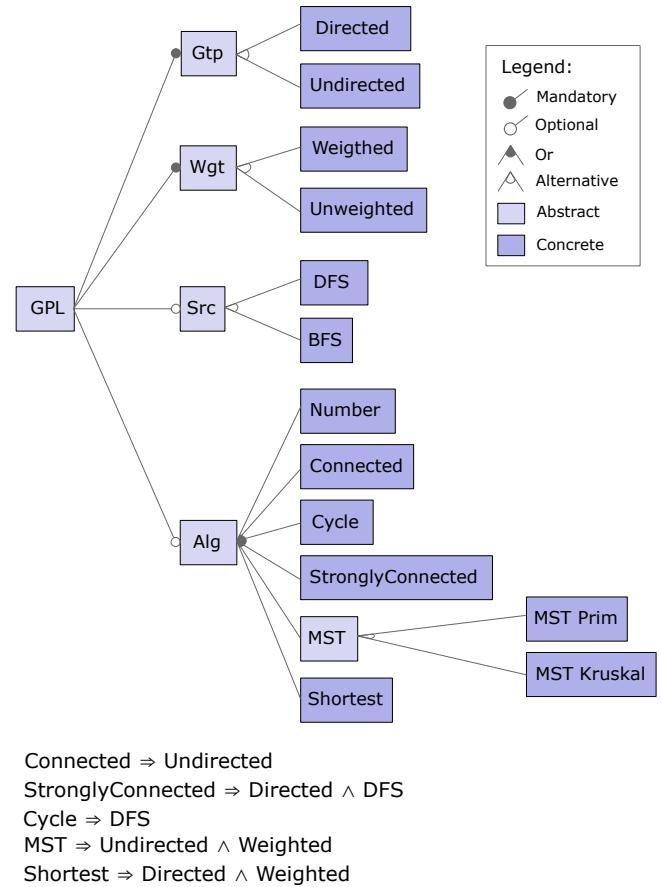
We challenge the research community to calculate samples, achieving a coverage measure for feature-, pair-, or three-wise coverage, with respect to our provided feature models. As solution for our challenge, we expect the calculated product sample as well as sampling statistics, containing CPU time and memory consumption. Furthermore, participants must provide, information about the achieved percentage of a coverage criteria (feature-, pair-, or three-wise). Beside those statistics, the sampling procedure used for the calculation must be provided as source code or executable file (Jar-file, Windows executable). For transparent comparison, we also expect a report about the procedure of sample generation, including a description of the used software libraries and hardware specification (e.g. operating system, memory size, and CPU).

To create a sample, participants are welcome to use already existing sampling algorithms and constraint solvers. In addition, we encourage all researchers to provide solutions containing newly developed sampling algorithms. All participants can choose the feature model(s) for which to generate samples. We provide a challenge category for every combination of feature model and coverage criterion, contained in submitted solutions. A solution can participate in any category for which it provides samples. Furthermore, a solution can participate in the challenge of sampling the whole product-line history, by generating samples for all versions contained in a product-line history. For both challenges, we consider as evaluation criteria the CPU time and memory consumption as well as the the sample size and the percentage of t-wise coverage for the given t. Hence, we ask all participants to provide those statistics for every generated sample in their solutions.

## 2 MOTIVATING EXAMPLE

In this section, we present a small example to illustrate basic concepts of feature modeling and sample creation. In addition, we illustrate the scalability challenge of sampling. As an example, an excerpt of the *Graph Product Line* (*GPL*), introduced by Lopez-Herrejon and Batory [16] is provided. The *Graph Product Line* represents a family of graph applications.

As shown in Fig. 1, our example includes 19 features in total, of which six are abstract and 13 are concrete. While abstract features



**Figure 1: Feature Model excerpt of the Graph Product Line**

are only used to group other features, concrete features are mapped to real implementation artifacts [28].

The combination of features is called product configuration. If a configuration conforms to all constraints, defined by the feature model, the configuration is valid. From valid product configurations real-world products are generated. Often abstract features are omitted in product configuration, because they do not represent actual implementation artifacts [28].

Features contained in our example are either organized as single feature or as a feature group. Single features can be mandatory (e.g. *Wgt*) or optional (e.g. *Src*). To build a valid configuration, a mandatory feature needs to be selected if its parent is selected. If the parent of an optional feature is selected, the optional feature itself does not need to be selected to build a valid product configuration.

The grouping of features describes their relation on the same hierarchy level. For example, the or-group formed by *Number*, *Connected*, *Cycle*, *StronglyConnected*, *MST*, and *Shortest*, defines that at least one of those features must be contained in a valid configuration. In contrast to or-groups, alternative-groups, require that exactly one feature must be selected, if the parent is selected. For example, *Edges* of a graph application can either be *Directed* or *Undirected*, but not both at the same time. Cross-tree constraints are propositional formulas over the features contained in a feature

<sup>1</sup><https://github.com/PettTo/Feature-Model-History-of-Linux>

<sup>2</sup>Linux evolution history from 2013-11-06 to 2018-01-14

<sup>3</sup>[https://github.com/PettTo/SPLC2019\\_The-Scalability-Challenge\\_Product-Lines/tree/master/Automotive02](https://github.com/PettTo/SPLC2019_The-Scalability-Challenge_Product-Lines/tree/master/Automotive02)

<sup>4</sup>[https://github.com/PettTo/SPLC2019\\_The-Scalability-Challenge\\_Product-Lines/tree/master/FinancialServices01](https://github.com/PettTo/SPLC2019_The-Scalability-Challenge_Product-Lines/tree/master/FinancialServices01)

**Table 1: Example for Coverage Criteria based on GPL**

	<b>t = 1</b>	<b>t = 2</b>	<b>t = 3</b>
Conf. 1	{Directed}	{Directed; Weighted}	{Directed; Weighted; DFS}
Conf. 2	{Undirected}	{Undirected; Weighted}	{Undirected; Weighted; DFS}
Conf. 3		{Directed; Unweighted}	{Directed; Unweighted; DFS}
Conf. 4		{Undirected; Unweighted}	{Undirected; Unweighted; DFS}
Conf. 5			{Directed; Weighted}
Conf. 6			{Undirected; Weighted}
Conf. 7			{Directed; Unweighted}
Conf. 8			{Undirected; Unweighted}

model. They are used to express dependencies between features [12]. For example, our feature model contains the dependency that a *Cycle* algorithm can only be selected, if *Depth-First-Search (DFS)* is selected as search strategy.

For real-world product lines, it is infeasible to test all valid product configurations. Often, a smaller subset of all possible configurations is used. The subset of configurations is called sample. To generate samples different procedures exist. Main concern of sampling is to achieve a sufficient test coverage with lower test effort (smaller test suits). Hence, sampling procedures try to produce a sample that is as small as possible w.r.t. the number of products. However, another important requirement is to be efficient regarding time, and memory consumption while retrieving a sample [30].

A promising approach to generate samples, which achieve a certain amount of test coverage is Combinatorial Interaction Testing (CIT). This approach is based on the assumption that most faults can be found considering interactions between a few features [17]. In Table 1, we show an example for the different CIT coverage criteria, based on our example. Each column of the table represents a minimal sample to achieve a certain feature interaction coverage. The first column of Table 1 shows a sample required to fulfill feature-wise coverage for *Directed*. To fulfill feature-wise ( $t = 1$ ) coverage, feature *Directed* must be selected and deselected in at least one configuration of a sample. The second column of Table 1 shows pair-wise ( $t = 2$ ) coverage for the features *Directed* and *Weighted*. As shown in Table 1, we must build all possible feature combinations (selection and deselection) between *Directed* and *Weighted* to cover pair-wise coverage. In the third column of Table 1, example configurations required to achieve three-wise ( $t = 3$ ) interaction coverage between *Directed*, *Weighted*, and *DFS* are shown. To fulfill three-wise coverage, all possible selection / deselection combinations between those three features are required.

Even though, pair-wise, and three-wise feature interaction coverage find more faults than feature-wise coverage, often they are not applicable for large product lines. The reason behind this, is the scalability problem of current sampling algorithms [2]. The sampling scalability challenge cannot be shown on a small example product line such as *GPL*. However, our own experience with sampling research shows that time and memory consumption as well as the size of calculated samples increase drastically for complex product lines.

### 3 SUBJECT PRODUCT LINES

For this challenge we provide a collection of feature models of three large product lines from different domains. For each product line we provide multiple feature models, each representing a part of the product-line history. Every provided feature model contains hundreds or thousands of features. Our collection of product lines contains two closed-source product lines from industrial application areas (Automotive02 and FinancialServices01) and one open-source product line (Linux kernel).

To protect sensitive data, all feature names of closed source product lines are obfuscated. In order to do so a hashing algorithm to replace feature names with unique IDs is used. To assure that a feature name gets the same unique ID throughout the evolution history, the hashing function is constant for all versions of the product line. Therefore, the obfuscation has no effect on the structure and dependencies of feature models and their evolution. In case of Automotive02, we received the feature models in an obfuscated form. For FinancialServices01 the FeatureIDE obfuscation algorithm was used. To access the algorithm we used the FeatureIDE library [13].

We provide the following feature models in our challenge:

**Linux.** With over 19,000 features, the Linux kernel is one of the largest software product lines currently available. All product-line versions of Linux are open-source and highly-configurable. The open-source character of the product line invites a huge community, which frequently extends and improves the kernel's functionality and its variability model. Because of those frequent updates, a long history of open-source variability models are available. The size and fast evolution of Linux poses challenges to the quality assurance of Linux. Hence, this product line is often used as benchmark to test new quality assurance procedures for product lines [1, 5, 6, 24, 26, 27].

Variability for the Linux kernel is managed in KConfig [31], which is a special file format created for the Linux kernel. To build a feature model for Linux the variability needs to be extracted from the KConfig files. We constructed a tool suite [21] based on KConfigReader [10, 14] and FeatureIDE library [13] to automatically extract Linux feature models from the Linux kernel Git repository<sup>5</sup>. KConfigReader is used to transform the variability model from KConfig into Conjunctive Normal Form (CNF) in the Dimacs format. During the transformation from KConfig to Dimacs format, KConfigReader performs the so called Tseitin transformation [29], to save computation time. Using the Tseitin transformation increases the number of literals contained in the CNF formulas. We transform the resulting Dimacs files into FeatureIDE XML format by using

<sup>5</sup><https://github.com/torvalds/linux>

**Table 2: Feature Model History: Automotive02**

Version Name	Date	Features	Constraints
V1	2015-04-01	14,010	666
V2	2015-05-01	17,742	914
V3	2015-06-01	18,434	1,300
V4	2015-07-01	18,616	1,369

**Table 3: Feature Model History: FinancialServices01**

Version Name	Date	Features	Constraints
2017-05-22	2017-05-22	557	1,001
2017-09-28	2017-09-28	704	1,136
2017-10-20	2017-10-20	712	1,142
2017-11-20	2017-11-20	711	1,148
2017-12-22	2017-12-22	716	1,148
2018-01-23	2018-01-23	712	1,028
2018-02-20	2018-02-20	759	1,034
2018-03-26	2018-03-26	771	1,080
2018-04-23	2018-04-23	774	1,079
2018-05-09	2018-05-09	771	1,080

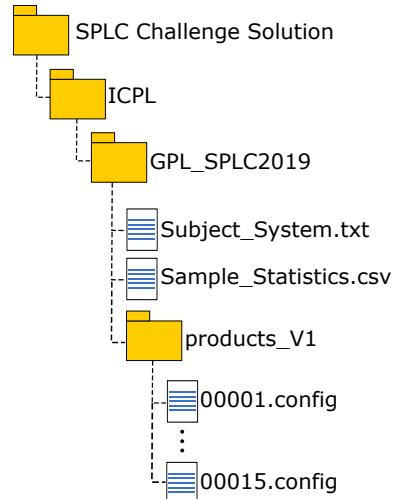
FeatureIDE import functionalities. As result, each of our provided Linux feature models contains over 60,000 features. This fact increases the scalability challenge for currently available sampling algorithms even more.

In this challenge, we offer a collection of feature models for 420 Linux versions as subject system for product sampling. Feature models of this collection are publicly available in our Git repository.<sup>6</sup> The feature models can be found in FeatureIDE XML and Dimacs format.

*Automotive02.* The Automotive02 product line was originally derived from the collaboration with our industrial partner from the automotive industry. We published those models and used them for the evaluation of feature model interfaces [22]. Later, the Automotive02 product line was used in different studies as subject product line from real-world applications [12, 20]. As shown in Table 2, we provide four feature models for this product line. They represent monthly snapshots from the evolution of Automotive02. The product line grew from 14,010 features and 666 constraints to a size of 18,616 features and 1,339 constraints, as shown in Table 2.

*FinancialServices01.* Originally, FinancialServices01 was obtained during a project in collaboration with an industrial partner from the financial services domain. We published and used the data to analyze feature-model anomalies [20]. We provide ten different versions of FinancialServices01 for our challenge. Each version represents a snapshot from the evolution history of the product line. Table 3 visualizes the statistics of each version. Throughout the evolution, FinancialServices01 grew from a product line with 557 features and 1,001 constraints to a product line with 771 features and 1,080 constraints.

<sup>6</sup><https://github.com/PettTo/Feature-Model-History-of-Linux>

**Figure 2: Example Folder Structure for Solutions**

## 4 EVALUATION OF SAMPLING ALGORITHMS

For this challenge, we provide the feature models and their history for Linux, Automotive02, and FinancialServices01. All the provided feature models can be used as subject system to generate product samples. We challenge our participants to calculate a sample for at least one version from at least one of the provided product lines. For the calculation of samples, any sampling technique can be used. We welcome any solution produced by already existing sampling procedures. In addition, all researchers are encouraged to present their own or even newly developed sampling techniques.

All feature models for this challenge are provided in FeatureIDE's XML format. This way, they can be processed with FeatureIDE [18]. This includes showing a graphical representation of the feature model, generating products with certain sampling algorithms, and transforming the XML representation into a supported file format (GUIDSL, SXFM, Velvet, or Dimacs).

Even though we recommend to use FeatureIDE to get started with this challenge, no participant is strictly bound to the usage of this tool.

To produce samples for our challenge any sampling technique can be used, even if the technique does not cover 100% of t-wise coverage for a certain value t. We expect that the sampling technique as well as all generated artifacts, such as the samples, sampling statistics, and a report about the subject system configuration (hardware and operating system), are publicly available. Only this way, we can assure that the results of different participants could be evaluated and compared against each other. Furthermore, we ask our participants to include a report and discussion about their experiences and lessons learned.

To encourage a uniform structure for submitting solutions, we provide a Git repository as an example structure.<sup>7</sup> The expected structure of this Git repository is visualized in Fig. 2. We expect folders for all algorithms used for sampling. The folder name should reflect the used algorithms. In our example case, this is the folder

<sup>7</sup>[https://github.com/PettTo/SPLC2019\\_The-Scalability-Challenge.git](https://github.com/PettTo/SPLC2019_The-Scalability-Challenge.git)

**Table 4: Solution Table for Graph Product Line**

Coverage Criterion (t)	Percentage Covered (%)	Sample Size	CPU Time (ms)	Memory Usage (KB)
2	100	15	2,000	161,000

*ICPL*. Inside the algorithm folder, we expect a separate folder for each product line, which was sampled. The folder must be named as the product line it contains samples for. We generated samples for our running example (*GPL*). Hence, our example folder is named *GPL\_SPLC2019*. Fig. 2 shows that we expect one *Experience Report*, one *Subject System Report*, one file containing sample statics in CSV format, and folders for the generated samples. Configurations of a submitted sample must be in FeatureIDE configuration format (\*.config). This format is equal to a text file containing each selected feature in a separate line, as shown in the example Git repository.<sup>8</sup>

The sample statistics must include the following artifacts:

- Name of the algorithm or solver library used for sampling
- Feature interaction coverage criterion (t)
- Percentage of achieved feature interaction coverage
- Number of products generated (sample size)
- CPU Time consumption needed to retrieve the sample
- Memory consumption needed to retrieve the sample (maximal heap usage, during sampling)

We expect that the sample statistics are provided as CSV file. Table 4 shows an example of how we expect the CSV structure of submitted solutions. The values contained in Table 4 are statistics for our *Graph Product Line* example. We used the ICPL algorithm integrated in FeatureIDE to create a sample that covers pair-wise feature interaction coverage. To get the sample size for example, we counted the configurations contain in the generated sample. To retrieve the CPU time, we implemented a method to measure the sampling duration into FeatureIDE. Regarding the values for memory consumption, we measured the maximal size of heap allocated during the sampling process using visualVM<sup>9</sup>. Furthermore, we measured the percentage of t-wise coverage reached, by using the algorithm shown in Listing 1.

```

1   function calculatePercentage(FM, T, S)
2     List TWC ← calculateAllTcombinations(FM, T);
3     int CC ← calculateCovertCombination(TWC,S);
4     return (CC * 100) / |TWC|;
5   end function

```

#### Listing 1: Calculation: Percentage of Coverage Achieved

As input for our function we expect a feature model (*FM*), a t-wise coverage (*T*), and a sample (*S*). In line two, a list of all feature combinations theoretically needed to achieve t-wise coverage is calculated (*TWC*). Thereafter, the number of feature combinations (*CC*) from *TWC* contained in sample *S* is calculated. Thenceforth, the percentage of covered feature combinations is returned.

<sup>8</sup>[https://github.com/PettTo/SPLC2019\\_The-Scalability-Challenge/tree/master/ICPL GPL\\_SPLC2019/products\\_v1](https://github.com/PettTo/SPLC2019_The-Scalability-Challenge/tree/master/ICPL GPL_SPLC2019/products_v1)

<sup>9</sup><https://visualvm.github.io/>

We expect that the CSV files provided as solutions conform to the following format:

- Use semicolon (";") as separator
- Use line feed ("\n") as line separator
- Provide a table header as shown in Table 4
- Use milliseconds (ms) as unit for sampling time
- Use kilobyte (KB) as unit for memory consumption
- Provide all numerical values as integer values

Submitted solutions will be evaluated based on the performance of their sampling algorithms. For the evaluation we use the provided values for sample size, CPU time consumption, memory consumption, and the percentage of t-wise coverage achieved. To establish a transparent evaluation, we will categorize the solutions based on which product line or product line version is used as subject system for the sampling process. Furthermore, we will consider, which kind of feature interaction coverage the solutions achieve. Another parameter for categorizing a solution is the subject system on which the sampling was performed. After categorizing the solution as described, sample size, memory usage, SPU time consumption, and the percentage of t-wise coverage achieved will be used as evaluation criteria. The aim for all participants should be to minimize sample size, memory consumption and CPU time consumption, while maximizing the percentage of achieved t-wise coverage.

Our participants can generate samples for all feature models of a product-line history to take part in a second challenge. In this challenge we evaluate how sampling techniques perform when applied to a product-line history. Even though every participant can use sampling techniques which sample each product-line version separately, we encourage our participants to actively develop sampling techniques, which consider the product line evolution explicitly. We ask our participants to provide the previously mentioned sample statics for each feature model of the product-line history separately. We will aggregate the values for each of our metrics. This way, we get one result for each metric, which represents the performance for sampling the whole product-line evolution history. Solutions will be evaluated based on this results.

## 5 SUMMARY

In this challenge, we presented the process of sampling large software product lines. Common sampling algorithms run into different scalability issues when performed on large systems. Those issues include running out of memory, needing to much computation time, and producing too large samples. We challenge the research community to generate samples for large product lines and submit them as solutions. As subject systems for this challenge, we provide feature models of three real-world product line histories. Solutions can be generated by using newly developed, or currently available sampling algorithms. Submitted solutions will be evaluated based on their performance. As evaluation criteria we use CPU time and memory consumption as well as sample size and the percentage of t-wise coverage achieved by the sample. Best performing algorithms minimize CPU time and memory consumption, and calculate a small sample which still achieves a high percentage of t-wise coverage. We will provide a comparison of the performance of all submitted solutions.

## REFERENCES

- [1] Iago Abal, Jean Melo, Stefan Stăniculescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *Trans. Software Engineering and Methodology (TOSEM)* 26, 3, Article 10 (2018), 10:1–10:34 pages.
- [2] Bestoun S. Ahmed, Kamal Z. Zamli, Wasif Afzal, and Miroslav Bures. 2017. Constrained Interaction Testing: A Systematic Literature Study. *IEEE Access* 5 (2017), 25706–25730.
- [3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.
- [4] Andrea Arcuri and Lionel Briand. 2011. Formal analysis of the probability of interaction fault detection using random testing. *TSE* 38, 5 (2011), 1088–1099.
- [5] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Understanding Linux Feature Distribution. In *Proc. of Workshop on Modularity in Systems Software (MISS)*. ACM, 15–20.
- [6] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2018. FEVER: An Approach to Analyze Feature-Oriented Changes and Artefact Co-Evolution in Highly Configurable Systems. *Empirical Software Engineering (EMSE)* 23, 2 (2018), 905–952.
- [7] Alireza Ensan, Ebrahim Bagheri, Mohsen Asadi, Dragan Gasevic, and Yevgen Biletskiy. 2011. Goal-Oriented Test Case Selection and Prioritization for Product Line Feature Models. In *Proc. Int'l Conf. on Information Technology: New Generations (ITNG)*. IEEE, 291–298.
- [8] Mats Grindal, Jeff Offutt, and Sten F Andler. 2005. Combination testing strategies: a survey. *STVR* 15, 3 (2005), 167–199.
- [9] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. Multi-Objective Test Generation for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 62–71.
- [10] Christian Kästner. 2017. Differential Testing for Variational Analyses: Experience from Developing KConfigReader. *arXiv preprint arXiv:1706.09357* (2017).
- [11] Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. 2011. Reducing Combinatorics in Testing Product Lines. In *Proc. Int'l Conf. on Aspect-Oriented Software Development (AOSD)*. ACM, 57–68.
- [12] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch Between Real-World Feature Models and Product-Line Research?. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 291–302.
- [13] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. 2017. FeatureIDE: Empowering Third-Party Developers. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 42–45.
- [14] Christian Kästner. 2018. KConfigReader. Website. Available online at <https://github.com/ckaestne/kconfigreader>; visited on August 29th, 2018.
- [15] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 81–91.
- [16] Roberto E. Lopez-Herrezon and Don Batory. 2001. A Standard Problem for Evaluating Product-Line Methodologies. In *Proc. Int'l Symposium on Generative and Component-Based Software Engineering (GCSE)*. Springer, 10–24.
- [17] Roberto E. Lopez-Herrezon, Stefan Fischer, Rudolf Ramler, and Aalexander Egyed. 2015. A First Systematic Mapping Study on Combinatorial Interaction Testing for Software Product Lines. In *Proc. Int'l Workshop on Combinatorial Testing (IWCT)*. IEEE, 1–10.
- [18] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.
- [19] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, Yguarata Cerqueira Cavalcanti, Eduardo Santana de Almeida, Vinicius Cardoso Garcia, and Silvio Romero de Lemos Meira. 2010. A regression testing approach for software product lines architectures. In *Software Components, Architectures and Reuse (SBARS), 2010 Fourth Brazilian Symposium on*. IEEE, 41–50.
- [20] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzeke. 2018. Anomaly Analyses for Feature-Model Evolution. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 188–201.
- [21] Tobias Pett. 2018. *Stability of Product Sampling under Product-Line Evolution*. Master's thesis. Braunschweig.
- [22] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2016. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 667–678.
- [23] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. 2012. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proc. Int'l Conf. on Fundamental Approaches to Software Engineering (FASE)*. Springer, 270–284.
- [24] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. 2007. Is the Linux Kernel a Software Product Line?. In *Proc. Int'l Workshop on Open Source Software and Product Lines (OSSPL)*. IEEE, 9–12.
- [25] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90,000 #Ifdefs Issue. In *Proc. USENIX Annual Technical Conference (ATC)*. USENIX Association, 421–432.
- [26] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proc. Europ. Conf. on Computer Systems (EuroSys)*. ACM, 47–60.
- [27] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2009. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 81–86.
- [28] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (2014), 6:1–6:45.
- [29] Grigori S Tseitin. 1983. On the complexity of derivation in propositional calculus. In *Automation of reasoning*. Springer, 466–483.
- [30] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 1–13.
- [31] Roman Zippel. 2017. KConfig Documentation. Website. Available online at <http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt>; visited on June 21st, 2019.

# ***t*-wise Coverage by Uniform Sampling**

Jeho Oh

jeho@cs.utexas.edu

University of Texas at Austin

Austin, Texas, USA

Paul Gazzillo

paul.gazzillo@ucf.edu

University of Central Florida

Orlando, Florida, USA

Don Batory

batory@cs.utexas.edu

University of Texas at Austin

Austin, Texas, USA

## ABSTRACT

Efficiently testing large configuration spaces of *Software Product Lines* (SPLs) needs a sampling algorithm that is both scalable and provides good *t*-wise coverage. The 2019 SPLC Sampling Challenge provides large real-world feature models and asks for a *t*-wise sampling algorithm that can work for those models.

We evaluated *t*-wise coverage by *uniform sampling* (US) the configurations of one of the provided feature models. US means that every (legal) configuration is equally likely to be selected. US yields statistically representative samples of a configuration space and can be used as a baseline to compare other sampling algorithms.

We used existing algorithm called Smarch to uniformly sample SPL configurations. While uniform sampling alone was not enough to produce 100% 1-wise and 2-wise coverage, we used standard probabilistic analysis to explain our experimental results and to conjecture how uniform sampling may enhance the scalability of existing *t*-wise sampling algorithms.

## CCS CONCEPTS

- Software and its engineering → Software product lines; • Theory of computation → Automated reasoning.

## KEYWORDS

software product lines, *t*-wise coverage, uniform sampling.

### ACM Reference Format:

Jeho Oh, Paul Gazzillo, and Don Batory. 2019. *t*-wise Coverage by Uniform Sampling. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3336294.3342359>

## 1 INTRODUCTION

*Software Product Lines* (SPLs) are highly configurable. Building blocks of SPL products are *features* that are increments of product functionality. Each product of an SPL is defined by a unique set of features called a *configuration*. A *feature model* declares each feature and constraints among features, so that a user can identify legal configurations with desired feature combinations [4]. As the number of features increase, the size of the *configuration space*, which is the set of all possible configurations, grows exponentially.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3342359>

A large configuration space could have over a trillion ( $>10^{12}$ ) configurations and is a challenge for testing, as testing every configuration is infeasible. Instead, prior work produced a small set of configurations to test selected features and their interactions. The aim is to get a ‘high’ *t*-wise coverage, ideally meaning 100% of *all* combinations of *t* features are covered by at least one configuration of the set. Achieving 100% can be infeasible for large spaces.<sup>1</sup> Common values for *t* include feature-wise (*t*=1), pair-wise (*t*=2), and three-wise coverage (*t*=3).

Different approaches start with a feature model and derive samples for *t*-wise coverage [1, 2, 6, 9, 10]. However, they do not scale well for many features and complex constraints, which limited their applicability to the real-world SPLs. Thus, the proposed Challenge [16] provides large real-world feature models and asks for a sampling algorithm that can generate configuration sets with good *t*-wise coverage for those models.

We explore *t*-wise coverage using *uniform sampling* (US) in this paper. US ensures that all configurations in a configuration space have equal probability of being selected, yielding a *statistically representative sample* of the space. US can be used as a baseline against which other sampling algorithms can compare as a benchmark [13].

Despite its utility, US for large SPLs was considered infeasible until recently [11, 13]. Prior work tried different methods to make sampling as random as possible, but none achieved US for large SPLs. We use a recently developed algorithm called Smarch [8], the first to perform US of configuration spaces of size  $10^{245}$ . Smarch utilizes a #SAT solver, which counts the number of solutions to a propositional formula [15]. We believe we are the first to explore *t*-wise coverage of US with probabilistic analyses to explain its coverage results.

Our contributions to the 2019 SPLC Sampling Challenge are:

- Demonstration of *t*-wise coverage that can be achieved by US; and
- Probabilistic analysis of configuration spaces that predicts the *t*-wise coverage by US and that may be useful for developing a practical *t*-wise sampling algorithm.

## 2 SMARCH: A US ALGORITHM

Smarch [8] is a US algorithm for SPLs based on a #SAT solver. Let  $\phi$  be the propositional formula of a feature model [3]. A #SAT solver can count the number of configurations in  $\phi$ 's configuration space, namely  $|\phi|$ . (Each solution to  $\phi$  is a configuration, and each configuration is a solution to  $\phi$ ). A #SAT solver extends a satisfiability solver by associating the number of solutions with each truth assignment [5]. Smarch uses sharpSAT [15], a state-of-the-art #SAT solver.

---

<sup>1</sup>Section 4 shows that uniform sampling alone will not provide 100% coverage unless the sample set is approximately the size of the configuration space.

Here is how Smarch achieves  $\mathbb{U}\mathbb{S}$ : A uniform random number generator can select an integer  $r$  in the range  $[1..|\phi|]$ . Smarch creates a one-to-one mapping that converts  $r$  into a unique configuration, so that  $\mathbb{U}\mathbb{S}$  of range  $[1..|\phi|]$  leads to  $\mathbb{U}\mathbb{S}$  of configurations.

Smarch recursively partitions  $\phi$  by a fixed order of variables to create a one-to-one mapping. A variable  $v \in \phi$  partitions  $\phi$  into disjoint spaces  $(\phi \wedge \neg v)$  and  $(\phi \wedge v)$ . #SAT can compute the number of solutions for each space, i.e.,  $|\phi \wedge \neg v|$  and  $|\phi \wedge v|$  respectively.

Then, for a random number  $r \in [1..|\phi|]$ , if  $r \leq |\phi \wedge \neg v|$  the  $(\phi \wedge \neg v)$  space is selected for recursive partitioning, otherwise  $(\phi \wedge v)$  is selected and  $|\phi \wedge \neg v|$  is subtracted from  $r$  to adjust the search in  $(\phi \wedge v)$ . This process is repeated for the next variable in  $\phi$ , until all variables are considered and a unique configuration has been determined.

Documenting scalability of Smarch and #SAT is beyond the scope of this paper – and is the subject of on-going work. Evidence in [8] reports Smarch is able to  $\mathbb{U}\mathbb{S}$  configuration spaces of size  $O(10^{248})$  whereas the nearest  $\mathbb{U}\mathbb{S}$  competitor's largest space is  $O(10^{13})$ .

### 3 EVALUATION

#### 3.1 Experimental Set-Up

Among the feature models provided in the Challenge, we used ‘FinancialServices01’ version ‘2018-05-09’. This feature model was given in FeatureIDE format [14], so we used the functionality of FeatureIDE to generate its propositional formula as a dimacs file. This file has 771 variables and 7,241 clauses. The size of the configuration space was determined to be  $9.7 \cdot 10^{14}$ , computed in a mere 46 milliseconds by sharpSAT [15].

We evaluated  $t$ -wise coverage for  $t=1$  and  $t=2$  and did the following to find valid combinations:

- (1) We derived a list of feature selections. With 771 features, there are  $771 \cdot 2 = 1,542$  possible selections since we consider both a feature and its negation;
- (2) We derived all possible 1-wise and 2-wise combinations from this list. 1-wise yields  $\binom{1542}{1} = 1542$  combinations and 2-wise yields  $\binom{1542}{2} = 1,188,111$ ; and
- (3) We filtered out invalid combinations using a SAT solver. If a combination is valid, the conjunction of the combination and the feature model should be satisfiable. For example, for a feature  $f$ , a 2-wise combination  $(f, \neg f)$  is invalid as these selections conflict with each other.

For 1-wise, 1,518 valid combinations were found (some features were mandatory). For 2-wise, 914,537 valid combinations were found.

We used Smarch to produce a  $\mathbb{U}\mathbb{S}$  set  $\mathbb{S}_n$  of  $n$  configurations. We have no idea what fraction of the valid combinations (computed above) are covered by  $\mathbb{S}_n$ . So we varied  $n$  to observe the results of increasing larger sets, using  $n = \{5, 10, 20, 30, 40, 50, 100, 200, 300, 400, 500, 1000, 1518\}$ . Then, for each set  $\mathbb{S}_n$ , we measured<sup>2</sup>:

- **Time taken to sample  $n$  configurations**, measured by the Linux ‘time’ tool;
- **Time taken to sample a configuration**, measured for each sample by Smarch;
- **Maximum memory used during sampling**, measured by the Linux ‘/usr/bin/time -v’ command; and

<sup>2</sup>The Challenge [16] explicitly requests sampling time and memory measurements.

- **$t$ -wise coverage for  $t=1$  and  $t=2$** , measured as the percentage of  $t$ -wise combinations covered in  $\mathbb{S}_n$ .

We conducted our evaluation on an Intel i7-6700@3.4Ghz Ubuntu 16.04 machine with 16GB of RAM. All the code and data for the evaluation are available at: [https://github.com/jeho-oh/Smarch\\_t\\_wise](https://github.com/jeho-oh/Smarch_t_wise).

#### 3.2 Experimental Results

Fig. 1a shows the total sampling time and Fig. 1b the time per sample. The X-axis is the number of samples ( $n$ ) and the Y-axis is the time in seconds. We observed:

- Total sampling time increases linearly with  $n$ ; and
- For all  $n$ , the average sampling time for a configuration was approximately 7 seconds, with standard deviation of 1 second. For all samples, the maximum sampling time was 10.1 seconds and the minimum was 3.5 seconds.
- The number of samples taken did not affect the time to sample a configuration.

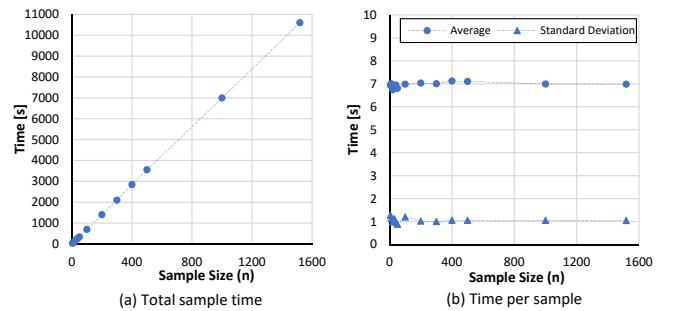


Figure 1: Sampling time.

Fig. 2 shows the maximum memory usage of Smarch, where the X-axis is the number of samples ( $n$ ) and the Y-axis is the memory size in megabytes. We observed:

- Maximum memory usage was stable, between 16.8MB and 17.1MB for all  $n$ ; and
- Sampling more configurations did not increase the maximum memory usage.

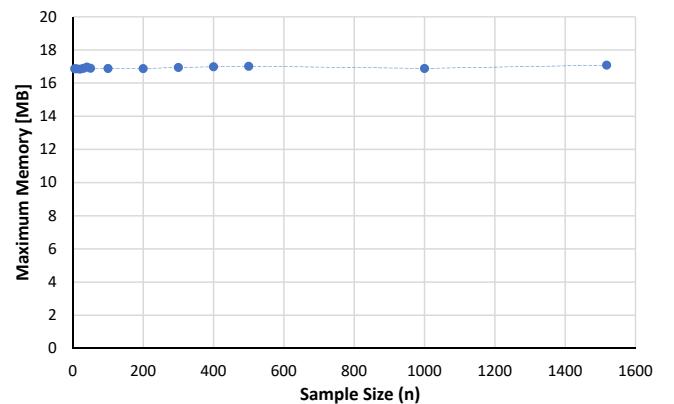


Figure 2: Maximum memory usage.

Fig. 3 shows the  $t$ -wise coverage result, where the X-axis is the number of samples ( $n$ ) and Y-axis is the percentage of the coverage.

Plots with different color indicates the results for different  $t$ . We observed:

- For all values of  $n$ , coverage for  $t=1$  was higher than  $t=2$ ;
- For  $\mathbb{S}_5$ , more than half of the feature combinations were covered for  $t=1$  and over 35% for  $t=2$ ;
- For both  $t=1$  and  $t=2$ , larger  $n$  yielded better coverage. With 1,518 samples, coverage for  $t=1$  was 61.7% and  $t=2$  was 47.6%;
- The difference in coverage between  $n=5$  and  $n=1,518$  was surprisingly small. For  $t=1$ , the difference was 6.4%. For  $t=2$ , the difference was 9.4%; and
- Although samples are expected to be statistically representative of the configuration space, their  $t$ -wise coverages seemed low. Both coverages improved imperceptibly for  $n \geq 200$ . Why this is so is explained in the next section.

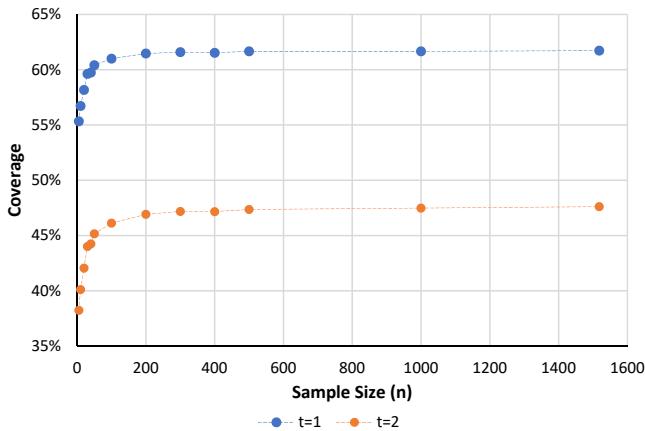


Figure 3:  $t$ -wise coverage.

We conclude that although  $\mathbb{U}\mathbb{S}$  is feasible with Smarch,  $\mathbb{U}\mathbb{S}$  alone is not enough to produce a 100%  $t$ -wise coverage.

## 4 ANALYSIS

$\mathbb{U}\mathbb{S}$  allows us to apply standard statistical analysis to explain our experimental results [7].

Let  $c$  denote a valid  $t$ -wise combination for a given  $t$ . Let  $v_c$  denote the fraction of all valid configurations that have  $c$  in the configuration space. Since every configuration has an equal probability of being selected by  $\mathbb{U}\mathbb{S}$ , the probability that a sample will have  $c$  is  $v_c$ .

$v_c$  can vary widely for different  $c$  because constraints among features may make certain combinations less frequent than others. A mandatory feature has  $v_c=1$  because it appears in all configurations. A feature with no constraints has  $v_c=0.5$ ; it can be freely enabled and disabled, making it appear in half of the valid configurations.

$v_c$  can be computed by a #SAT solver. Let  $\phi$  be the propositional formula of an SPL's feature model. Let  $\phi_c$  be the propositional formula of the conjunction of  $c$ 's features. We can use a #SAT solver to compute  $v_c$  as:

$$v_c = \frac{|\phi_c|}{|\phi|} \quad (1)$$

The probability  $p(c, n)$  that at least one of  $n$  samples includes combination  $c$  is:

$$p(c, n) = 1 - (1 - v_c)^n \quad (2)$$

where the more samples taken, the higher the probability we encounter combination  $c$ . The value of  $p(c, n)$  largely depends on how often this combination appears in the configuration space, i.e.,  $v_c$ .

In our experiments of the previous section, we discovered:

- 61.5% of all 1-wise combinations have a ratio  $v_c > 0.9$ . Even with the minimum number of samples we used in the evaluation ( $n=5$ ), these combinations have more than 0.99 probability of being encountered in  $n=5$  samples; and
- 38.8% of the 1-wise combinations have a ratio of  $v_c < 0.0001$ . Even with the maximum number of samples we used in the evaluation ( $n=1815$ ), they have less than 0.15 probability of being encountered in  $n=1815$  samples.

We can use  $p(c, n)$  to predict  $t$ -wise coverage. Let  $\mathbb{C}_t$  denote the set of all valid  $t$ -wise combinations, where  $|\mathbb{C}_t|$  is the number of combinations in  $\mathbb{C}_t$ . The estimated  $t$ -wise coverage  $E(\mathbb{C}_t, n)$  for a given  $t$ ,  $n$  is:

$$E(\mathbb{C}_t, n) = \frac{1}{|\mathbb{C}_t|} \sum_{c \in \mathbb{C}_t} p(c, n) = \frac{1}{|\mathbb{C}_t|} \sum_{c \in \mathbb{C}_t} (1 - (1 - v_c)^n) \quad (3)$$

Fig. 4 uses (blue) X markers to plot  $E(\mathbb{C}_1, n)$  and (brown) X markers for  $E(\mathbb{C}_2, n)$  with our experimental results (● for  $t=1$  and ● for  $t=2$ ) overlaid. Eqn. (3) accurately predicts the results of our experiments and also explains why the coverage of  $t=1$  is higher than that for  $t=2$ : there are many 2-way feature combinations ( $c_{ij}$ ) that are much less likely than any 1-way combination ( $c_k$ ), meaning  $v_{c_k} \gg v_{c_{ij}}$ .

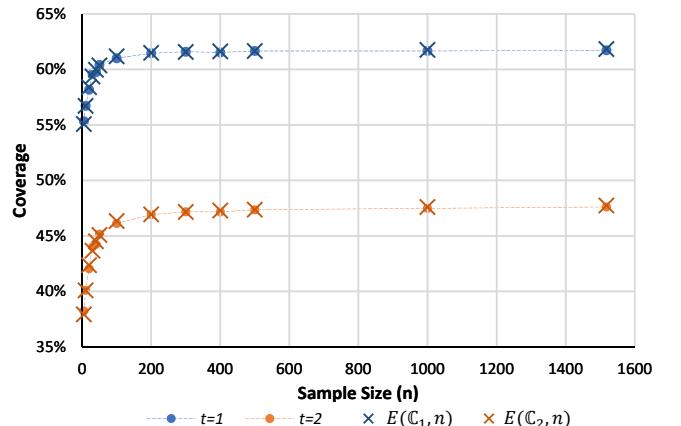


Figure 4:  $t$ -wise coverage estimation.

It is interesting to explore the relationship between coverage and larger sample set sizes which are infeasible to explore experimentally. Fig. 5 shows the estimated  $t$ -wise coverage for  $n$  up to  $10^{14}$ , which is approximately 10% of the configuration space (ie.,  $9.7 \times 10^{14}$ ). We observed:

- With  $10^{14}$  samples, more than 99.99% of 1-wise and 2-wise combinations are expected to be covered. Of course, this is almost enumeration; and

- Many combinations will be covered with a small number of samples, over 30% of 2-way combinations are not likely to be covered even with  $10^7$  samples(!).

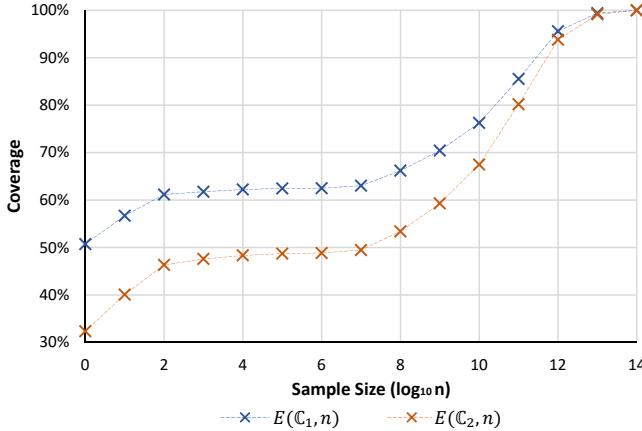


Figure 5:  $t$ -wise coverage estimation for large  $n$ .

We could accurately predict these results because Smarch can uniformly sample from a configuration space and standard probabilistic analyses rely on  $\text{US}$  [7].

Our analysis suggests possible enhancements to existing  $t$ -wise approaches. Once  $v_c$  values are known, we can determine which combinations can be covered by a small number of  $\text{US}$ s. Then, for combinations that are unlikely to be found by  $\text{US}$ , we may either: 1) constrict the configuration space with constraints to (recursively) sample configuration sub-space of interest [12] or 2) use existing approaches that do not rely on  $\text{US}$ . As sampling with many features limits the scalability of existing approaches,  $\text{US}$  may improve sampling scalability by reducing the features to consider. An equally important issue is to define a reasonable  $t$ -wise coverage (percentage) for large configuration spaces (other than 100%) for practitioners to use.

## 5 CONCLUSIONS AND FUTURE WORK

As  $\text{US}$  of configurations was considered infeasible, probabilistic analyses of a configuration space based on  $\text{US}$  was unexplored or considered unexplorable. We used a recently developed algorithm, Smarch [8], to  $\text{US}$  configurations of a configuration space. We also derived probabilistic models to explain Smarch results. We showed:

- $\text{US}$  alone is **not** enough to produce 100%  $t$ -wise coverage; and
- Distribution of  $v_c$  can be used to predict the  $t$ -wise coverage of  $\text{US}$ .

Our work opens new possibilities on analyzing an SPL configuration space and deriving samples for testing. As  $\text{US}$  produces statistically representative samples of a configuration space, it may be possible to utilize the information from samples to improve the efficiency of existing approaches. As a future work, we plan to:

- Analyze other systems to validate and expand our insights on probabilistic analyses;
- Derive an algorithm utilizing  $\text{US}$  for  $t$ -wise coverage; and
- Enhance the performance of the Smarch algorithm.

## ACKNOWLEDGMENTS

Work by Gazzillo is supported by NSF CCF-1840934. Work by Oh and Batory is supported by NSF grant CCF-1421211.

## REFERENCES

- [1] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. InCLing: efficient product-line testing using incremental pairwise sampling. In *ACM SIGPLAN Notices*. ACM, ACM, New York, NY, USA, 144–155.
- [2] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. 2019. Effective product-line testing using similarity-based product prioritization. *Software & Systems Modeling* 18, 1 (2019), 499–521.
- [3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer, Berlin, Heidelberg.
- [4] Don Batory. 2005. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*. Springer, Springer, NY, USA, 7–20.
- [5] Armin Biere, Marijn Heule, and Hans van Maaren. 2009. *Handbook of satisfiability*. Vol. 185. IOS press, IEEE.
- [6] Brady J Garvin, Myra B Cohen, and Matthew B Dwyer. 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering* 16, 1 (2011), 61–102.
- [7] C.M. Grinstead and J.L. Snell. 2019. Intro to Probability - Dartmouth College. [https://www.dartmouth.edu/~chance/teaching\\_aids/books\\_articles/probability\\_book/amsbook.mac.pdf](https://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/amsbook.mac.pdf).
- [8] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. 2019. *Uniform Sampling from Kconfig Feature Models*. Technical Report TR-19-02. University of Texas at Austin, Department of Computer Science.
- [9] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of realistic feature models make combinatorial testing of product lines feasible. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, Springer, Berlin, Heidelberg, 638–652.
- [10] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. 2012. Generating better partial covering arrays by modeling weights on sub-product lines. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, Springer, Berlin, Heidelberg, 269–284.
- [11] Christian Kaltenacker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-Based Sampling of Software Configuration Spaces. In *Proceedings of the 2019 International Conference on Software Engineering*. ICSE, IEEE/ACM, Piscataway, NJ, USA, 0.
- [12] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, IEEE/ACM, Piscataway, NJ, USA, 61–71.
- [13] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroye, and Maxime Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *12th International Conference on Software Testing, Verification, and Validation ICST 2019*. IEEE, Piscataway, NJ, USA.
- [14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85.
- [15] Marc Thurley. 2006. sharpSAT—counting models with advanced component caching and implicit BCP. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 424–429.
- [16] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *Software Product Line Conference, Challenge Case*. ACM, ACM, New York, NY, USA.

# A Graph-Based Feature Location Approach Using Set Theory

Richard Müller

Leipzig University

Leipzig, Germany

rmueller@wifa.uni-leipzig.de

Ulrich Eisenecker

Leipzig University

Leipzig, Germany

eisenecker@wifa.uni-leipzig.de

## ABSTRACT

The ArgoUML SPL benchmark addresses feature location in Software Product Lines (SPLs), where single features as well as feature combinations and feature negations have to be identified. We present a solution for this challenge using a graph-based approach and set theory. The results are promising. Set theory allows to exactly define which parts of feature locations can be computed and which precision and which recall can be achieved. This has to be complemented by a reliable identification of feature-dependent class and method traces as well as refinements. The application of our solution to one scenario of the benchmark supports this claim.

## CCS CONCEPTS

- Software and its engineering → Software product lines; Software reverse engineering;
- Information systems → Graph-based database models.

## KEYWORDS

Feature location, Software Product Lines, Benchmark, Reverse Engineering, Extractive Software Product Line Adoption, ArgoUML, Set theory, Static analysis, Graph database, Neo4j, Cypher, jQAssistant

### ACM Reference Format:

Richard Müller and Ulrich Eisenecker. 2019. A Graph-Based Feature Location Approach Using Set Theory. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/3336294.3342358>

## 1 INTRODUCTION

Feature location techniques aim at identifying a mapping from features to source code elements. A feature is “*a prominent or distinctive and user visible aspect, quality, or characteristic of a software system or systems*” [2]. Locating features plays an important role during software maintenance of single systems [1] and of Software Product Lines (SPLs) [5]. In case of SPLs, single features as well as feature combinations and feature negations have to be identified.

Martinez et al. [3] proclaim the ArgoUML SPL benchmark to foster research in feature location for SPLs. It provides a list of features with names and descriptions, a set of scenarios each containing a set of ArgoUML variants, a common ground-truth, and a Java

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3342358>

program to automatically calculate precision, recall, and F1-score based on the feature location results and the ground-truth. The challenge is to develop a technique that locates single features, feature combinations, and feature negations in 15 different scenarios derived from the ArgoUML SPL.

The contribution of this paper is a solution for this challenge. According to the feature location taxonomy of Dit et al. [1], we use a *static analysis* taking Java source code artifacts as *input* and extracting a software graph for each ArgoUML variant. The software graphs act as *data sources*. These graphs are created with jQAssistant, a tool that scans software artifacts and stores them as graphs in a Neo4j database.<sup>1,2</sup> It can be extended with plugins to support certain types of software artifacts [4]. For this challenge, we use the Java source code plugin.<sup>3</sup> Next, a trace graph based on the software graphs of the given scenario is created using the graph query language Cypher.<sup>4</sup> We use set theory to define all elementary subsets for the features and their combinations specific to the given scenario. Based on this, feature traces are computed for the actually existing feature-specific subsets using the trace graph. The *output* is a text file for each feature with class and method traces. The technique is *evaluated* with the traditional scenario of the ArgoUML SPL benchmark [3].

## 2 APPROACH

Our proposed solution comprises a conceptual and a technical part. In the first part, we map the given problem to set theory which facilitates finding a principal solution. In the second part, we use the graph database Neo4j and its query language Cypher, which provide a proper representation for sets of traces and methods for set operations.

### 2.1 Concept

First, let us introduce the core system  $S_0$  of the SPL  $S$ .  $S_0$  implements only the core functionality, but no additional feature. Let  $T_0$  be the set of software traces used to implement  $S_0$ . Next, we assume that there are three systems  $S_1$ ,  $S_2$ , and  $S_3$ , also instances of  $S$ .  $S_1$  implements both features  $F_1$  and  $F_2$ .  $S_2$  solely implements feature  $F_1$  and  $S_3$  solely  $F_2$ .  $T_1$  is the set of software traces implementing  $S_1$ . The same applies to  $T_2$  and  $S_2$  as well as  $T_3$  and  $S_3$ . Table 1 summarizes these sets.

We denote the set  $F_{1,pure}$  as exclusive traces required for  $F_1$ , but no other feature. The same applies to  $F_{2,pure}$  and  $F_2$ . In the case of two features  $F_1$  and  $F_2$ , we have to care for two special cases.

---

<sup>1</sup><https://jqassistant.org/>

<sup>2</sup><https://neo4j.com/>

<sup>3</sup><https://github.com/softvis-research/jqa-javasrc-plugin/tree/splc-challenge>

<sup>4</sup><https://www.opencypher.org>

**Table 1: Relationships between sets of system, implemented features, and software traces with two features**

System	Features	Traces
$S_0$	<i>none</i>	$T_0$
$S_1$	$F_1, F_2$	$T_1$
$S_2$	$F_1$	$T_2$
$S_3$	$F_2$	$T_3$

First, there may be software traces, which are common to both features, that is, traces are included for the condition `//#if defined(F1) or //#if defined(F2)`. This set is denoted as  $F_{1\text{or}2}$ . Now, we can determine the following sets.

$$F_1 := F_{1\text{pure}} \cup F_{1\text{or}2} := T_2 \setminus T_0 \quad (1)$$

$$F_2 := F_{2\text{pure}} \cup F_{1\text{or}2} := T_3 \setminus T_0 \quad (2)$$

$F_1$  contains all software traces, which are relevant for this feature including exclusive (*pure*) and common (*or*) traces. The same applies to  $F_2$ . The operator  $\cup$  denotes the union of two sets and the operator  $\setminus$  denotes the result of diminishing the left set by the right set.

Second, there may be software traces, which are only included if both features are present, that is, traces are included for the condition `//#if defined(F1) and //#if defined(F2)`. This set is denoted as  $F_{1\text{and}2}$ . Please note, that  $F_{1\text{and}2}$  only exists if both features,  $F_1$  and  $F_2$ , are present; otherwise it is non-existent.

$$F_{1\text{pure}} \cup F_{1\text{and}2} := (T_1 \setminus T_0) \setminus T_3 \quad (3)$$

$$F_{2\text{pure}} \cup F_{1\text{and}2} := (T_1 \setminus T_0) \setminus T_2 \quad (4)$$

$$F_{1\text{and}2} := (F_{1\text{pure}} \cup F_{1\text{and}2}) \cap (F_{2\text{pure}} \cup F_{1\text{and}2}) \quad (5)$$

$T_1 \setminus T_0$  contains all software traces of all features without the core traces. If we subtract the set  $T_3$  from this set we get the exclusive (*pure*) traces for feature  $F_1$  and the traces if both features  $F_1$  and  $F_2$  are present (*and*). The same applies to  $T_2$  for  $F_2$ . The intersection  $\cap$  of both sets results in the traces of the feature combination  $F_{1\text{and}2}$ .

Given this, the sets  $T_0$ ,  $F_{1\text{pure}}$ ,  $F_{2\text{pure}}$ ,  $F_{1\text{or}2}$ , and  $F_{1\text{and}2}$  are a complete dissection of the software system  $S_1$ , implementing both features  $F_1$  and  $F_2$ . *Complete dissection* means that each possible intersection of two sets from the index set with elements  $T_0$ ,  $F_{1\text{pure}}$ ,  $F_{2\text{pure}}$ ,  $F_{1\text{or}2}$ , and  $F_{1\text{and}2}$  is empty. Furthermore, no software trace exists, which is not an element of one of the aforementioned sets.

The preceding approach can be generalized to any number of features belonging to a software product line and the corresponding sets. We abstain from doing that because of space restrictions.

Now, the problem of locating features in a given scenario of the ArgoUML SPL benchmark can be solved by computing the sets of feature-specific traces depending on the definition of the corresponding elementary feature subsets. When writing the initial version of this paper, we had to define the elementary feature

subsets manually. Meanwhile, we designed an appropriate algorithm and implemented it, which automates feature location in the context of the ArgoUML SPL benchmark.<sup>5</sup>

Because of initially missing an adequate algorithm, we manually determined the elementary feature subsets of the traditional scenario. This scenario has 10 variants where one system is without all the optional features, one is with all the features, and then, for each feature, one system with all the features enabled and this feature disabled [3]. Table 2 summarizes the given sets.

**Table 2: Relationships between sets of system, implemented features, and software traces in the traditional scenario**

System	Features	Traces
$S_0$	<i>none</i>	$T_0$
$S_1$	$F_1, F_2, F_3, F_4, F_5, F_6, F_7, F_8$	$T_1$
$S_2$	$F_2, F_3, F_4, F_5, F_6, F_7, F_8$	$T_2$
$S_3$	$F_1, F_3, F_4, F_5, F_6, F_7, F_8$	$T_3$
$S_4$	$F_1, F_2, F_4, F_5, F_6, F_7, F_8$	$T_4$
$S_5$	$F_1, F_2, F_3, F_5, F_6, F_7, F_8$	$T_5$
$S_6$	$F_1, F_2, F_3, F_4, F_6, F_7, F_8$	$T_6$
$S_7$	$F_1, F_2, F_3, F_4, F_5, F_7, F_8$	$T_7$
$S_8$	$F_1, F_2, F_3, F_4, F_5, F_6, F_8$	$T_8$
$S_9$	$F_1, F_2, F_3, F_4, F_5, F_6, F_7$	$T_9$

Due to the given configurations, it is not possible to determine the sets of common (*or*) feature traces in the traditional scenario. However, we can determine exclusive (*pure*) feature traces and feature combination (*and*) traces. For example, to get the exclusive traces for  $F_1$  and  $F_2$  as well as the traces for the feature combination  $F_{1\text{and}2}$ , we apply the following set operations.

$$F_{1\text{pure}} := ((T_1 \setminus T_0) \setminus T_2) \setminus \bigcup_{i=1}^8 \bigcup_{j=i+1}^8 F_{i\text{and}j} \quad (6)$$

$$F_{2\text{pure}} := ((T_1 \setminus T_0) \setminus T_3) \setminus \bigcup_{i=1}^8 \bigcup_{j=i+1}^8 F_{i\text{and}j} \quad (7)$$

$$F_{1\text{and}2} := ((T_1 \setminus T_0) \setminus T_2) \cap ((T_1 \setminus T_0) \setminus T_3) \quad (8)$$

To get the traces of the other features  $F_3$  to  $F_8$  and feature combinations these operations are repeated analogously. Computing these sets we expect a precision of 1.0. The recall will be lower because we cannot determine the sets of common (*or*) traces for two or more features in this scenario. Thus, they will result in values of 0.0, for both, precision and recall. While this may sound disappointing at the first look, it precisely reflects the maximum information, which can be computed (provided the identification of relevant software traces is perfect) for the given information. Hence, the described procedure can be considered as a principle approach for feature location.

<sup>5</sup><https://github.com/softvis-research/featurelocation>

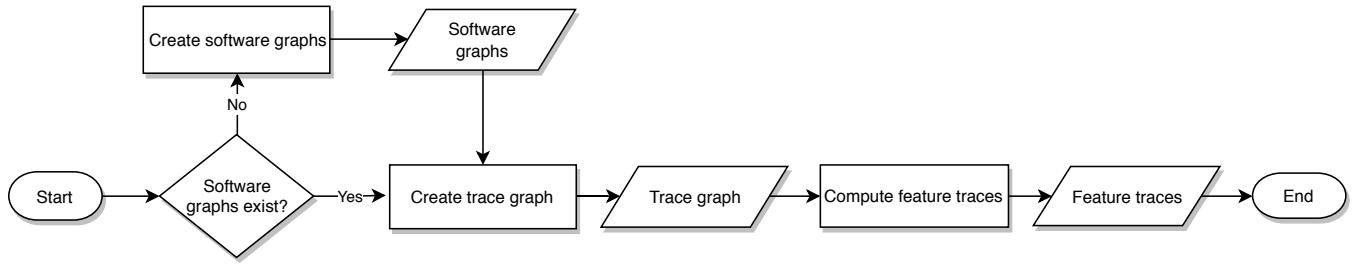


Figure 1: Flowchart of the graph-based feature location approach using set theory

## 2.2 Implementation

We have implemented the feature location technique in Java using the provided Eclipse workspace. The source code is available on GitHub.<sup>6</sup> The flowchart in Figure 1 shows the process of applying the technique.

For each variant it is checked whether a corresponding software graph exists. If not, it is created by the command-line version of jQAssistant using the Java source code plugin. Thus, each variant is scanned exactly once and the resulting software graph is stored in an embedded Neo4j database. Figure 2 shows a part of the software graph including the nodes `Type` and `Method`, the relationship `DECLARES` as well as node properties `name`, `fqn` (fully qualified name), and `signature`.

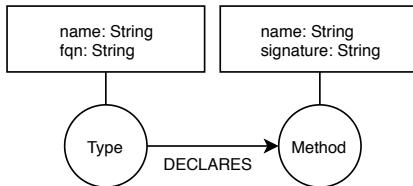


Figure 2: Part of the software graph created for each variant

Listing 1 shows the corresponding Cypher queries for getting the traces to complete classes and methods. The first query returns all types whose `fqn` starts with `org.argouml`, that are not inner or anonymous classes, and where the name consists of more than one letter. The second query returns all methods whose declaring type meets the above mentioned requirements.

### Listing 1: Cypher queries for getting the traces to complete classes and methods

```

MATCH (type:Type)
WHERE type.fqn STARTS WITH 'org.argouml'
AND NOT (:Type)-[:DECLARES]->(type) AND NOT type.fqn CONTAINS '$'
AND NOT type.fqn CONTAINS 'Anonymous' AND NOT size(type.name) = 1
RETURN DISTINCT type.fqn AS type

MATCH (type:Type)-[:DECLARES]->(method:Method)
// same conditions as above
RETURN DISTINCT type.fqn AS type, method.name AS method, method.
signature AS signature
  
```

<sup>6</sup><https://github.com/softvis-research/argouml-spl-benchmark>

Based on these queries the trace graph for the scenario is created. Figure 3 shows this graph including the nodes `Feature`, `Configuration`, `Trace:Class`, and `Trace:Method`, the relationships `HAS` and `DECLARES` as well as their properties `name` and `value`. For each configuration the active features are added as `Feature`, the queried types are added as traces labeled `Trace:Class` and the queried methods are added as traces labeled `Trace:Method` to the trace graph.

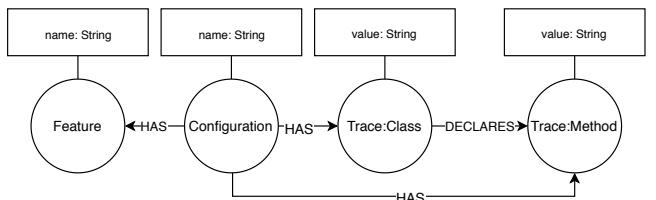


Figure 3: The trace graph created for a scenario

Now, we can apply set operations on the trace graph using Cypher including *Awesome Procedures On Cypher* (APOC).<sup>7</sup> Please note, that this implementation is specific to the traditional scenario. Other scenario sets have to be computed differently.

Listing 2 shows the Cypher queries for the necessary set operations to compute the sets for the exclusive traces  $F_{1^{pure}}$  to  $F_{8^{pure}}$  and the feature combinations. First, we query all traces from the configuration where all features are disabled and label them with `core`. These traces correspond to  $T_0$ . Second, we query  $T_1$  from the configuration where all features are enabled and apply the set difference  $T_1 \setminus T_0$  using the APOC procedure `apoc.coll.subtract()` and label the resulting traces with `FeatureTrace`. Third, we query  $T_2$ , the traces where  $F_1$  is disabled and apply the same APOC procedure to apply the set difference  $(T_1 \setminus T_0) \setminus T_2$  resulting in the traces of  $F_{1^{pure}}$  and of all feature combinations. For each identified trace a relationship between `Feature` and `Trace` is created and its value is set to `pure`. These operations are repeated for all other features  $F_2$  to  $F_8$ . Fourth, we query all traces of each feature and apply the APOC procedure `apoc.coll.intersection()` on every pair of features. For every identified feature combination relationships are created and their value is set to `and`. If there is a relationship with the value `pure` it is deleted. Finally, the identified traces are written to a text file for each feature and feature combination.

<sup>7</sup><https://neo4j-contrib.github.io/neo4j-apoc-procedures/>

### Listing 2: Cypher queries to compute feature traces

```

MATCH (c:Configuration)-[:HAS]->(ct:Trace)
WHERE c.name = 'P01_AllDisabled.config' SET ct:Core
// ...
MATCH (c:Configuration)-[:HAS]->(fct:Trace)
WHERE c.name = 'P02_AllEnabled.config'
WITH cts, collect(fct) AS fcts
WITH apoc.coll.subtract(fcts, cts) AS fts
FOREACH (f IN fts | SET f:FeatureTrace )
//...
MATCH (notf:Feature)<-[:HAS]->-(notft:FeatureTrace)
//...
WITH apoc.coll.subtract(allfts, notfts) AS fts
//...
FOREACH (ft IN (ffts) | CREATE (f)-[:HAS{value:'pure'}]->(ft))
//...
MATCH (f1:Feature)-[:HAS]->(dt:FeatureTrace)
MATCH (f1)-[:HAS]->(:FeatureTrace)-[:DECLARES]->(idt)
//...
WITH f1, f2, apoc.coll.intersection(fts1, fts2) AS andts
//...
MERGE (f1)-[:HAS{value:'and'}]->(fandt)
MERGE (f2)-[:HAS{value:'and'}]->(fandt) DELETE r1 DELETE r2

```

## 3 EVALUATION

We have located traces to complete classes and methods for 12 out of 24 features and feature combinations in the traditional scenario. The benchmark metrics are summarized in Table 3. The underlying ground-truth is pruned, that is, all traces of class and method refinements have been removed.

**Table 3: Benchmark metrics for 12 of 24 features and feature combinations in the traditional scenario based on a pruned ground-truth without refinement traces**

Name	Precision	Recall	F1-score
ACTIVITYDIAGRAM	1.00	0.49	0.66
ACTIVITYDIAGRAM_and_STATEDIAGRAM	1.00	1.00	1.00
COGNITIVE	1.00	1.00	1.00
COGNITIVE_and_DEPLOYMENTDIAGRAM	1.00	0.93	0.96
COGNITIVE_and_SEQUENCEDIAGRAM	1.00	1.00	1.00
COLLABORATIONDIAGRAM	1.00	0.95	0.98
COLLABORATIONDIAGRAM_and_SEQUENCEDIAGRAM	1.00	1.00	1.00
DEPLOYMENTDIAGRAM	1.00	1.00	1.00
LOGGING	1.00	0.67	0.80
SEQUENCEDIAGRAM	1.00	0.96	0.98
STATEDIAGRAM	1.00	0.63	0.77
USECASEDIAGRAM	1.00	1.00	1.00
Average	1.00	0.89	0.93

As expected, we have identified all exclusive (*pure*) traces of single features and all traces of feature combinations (*and*). The recall values below 1.0 are due to the missing common (*or*) traces as described in Section 2.1. The remaining 12 features, feature combinations, and feature negations have a precision and recall of 0.0 because they only contain traces of class and method refinements that are currently not available in the software graph.

We have measured the time for creating the software graphs and the trace graph on a Zotac MAGNUS EN1070 (Intel Core i5-6400T, 16 GB RAM) with Windows 10 as operating system and a Java development toolkit in version 12. In the traditional scenario, it takes 60min (3606903ms) to create the 10 software graphs and 45s (44987ms) to create the trace graph and compute the feature traces.

## 4 DISCUSSION

Next, we discuss the strengths and weaknesses of the feature location technique. We will suggest possible improvements mitigating the mentioned weaknesses in Section 5.

### 4.1 Strengths

The proposed approach locates features, feature combinations, and feature negations with a precision of 1.0 and a recall of 1.0 provided, the required sets of software traces are available. We demonstrated that for exclusive feature traces and traces of feature combinations to complete classes and methods in the traditional scenario.

Furthermore, the implementation of the technique is very compact. This is mainly due to the sensible integration of existing tools, such as jQAssistant and Neo4j.

### 4.2 Weaknesses

Until recently, we had to determine ourselves which elementary feature subsets can be computed for a given scenario and which questions can be answered accordingly. It is obvious that for the scenario with all possible variants, that is, 256, all the information is given to compute any possible subset.

The software graphs are created with the Java source code plugin for jQAssistant. At the moment this plugin does not consider imports, fields, and method statements. For these reasons, only traces to complete classes and methods are supported.

Moreover, the creation of the software graph for one variant takes approximately 6 minutes. In case of the scenario with 256 variants, the scan process to create the software graphs takes almost 26 hours.

## 5 CONCLUSION AND FUTURE WORK

In this paper we outline a principal solution to the feature location benchmark with ArgoUML SPL [3]. We have mapped the given problem to set theory and developed a graph-based solution for the traditional scenario with Java, Neo4j, and Cypher. Our technique locates exclusive traces and pair-wise feature interaction traces to complete classes and methods of 12 out of 24 features and feature combinations with a precision of 1.0 and a recall of 1.0.

After extending the Java source code plugin for jQAssistant to scan imports, fields, and method statements, it will be possible to detect class and method refinements for locating the remaining features, feature combinations, and feature negations. Including the implementation of the algorithm for defining all elementary feature subsets for a given scenario will allow to automate feature location in the context of the ArgoUML SPL benchmark. Furthermore, performance bottlenecks have to be identified and replaced by adequate optimizations. Finally, if a specific subset cannot be exactly computed because of missing necessary variants in a given scenario, alternative techniques to identify feature-specific traces could be applied, for example, feature names as substrings of given identifiers.

## ACKNOWLEDGMENTS

We especially acknowledge the work of Dirk Mahler, the main developer jQAssistant, and Michael Hunger, the main developer of APOC.

## REFERENCES

- [1] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *J. Softw. Evol. Process* 25, 1 (2013), 53–95. <https://doi.org/10.1002/smrv.567>
- [2] Kang K. C., Sholom G Cohen, James A Hess, William E Novak, A Spencer Peterson, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feasibility Study Feature-Oriented Domain Analysis (FODA)*. Technical Report November. Carnegie-Mellon University Software Engineering Institute.
- [3] Jaber Martinez, Nicolas Ordoñez, Xhevahire Térnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. 2018. Feature Location Benchmark with argoUML SPL. In *Proc. 22Nd Int. Syst. Softw. Prod. Line Conf. - Vol. 1 (SPLC '18)*. ACM, New York, NY, USA, 257–263. <https://doi.org/10.1145/3233027.3236402>
- [4] Richard Müller, Dirk Mahler, Michael Hunger, Jens Nerche, and Markus Harrer. 2018. Towards an Open Source Stack to Create a Unified Data Source for Software Analysis and Visualization. In *Proc. 6th IEEE Work. Conf. Softw. Vis.* IEEE, Madrid, Spain. <https://doi.org/10.1109/VISSOFT.2018.00019>
- [5] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Eng. Prod. Lines, Lang. Concept. Model.* 29–58.

# Comparison-Based Feature Location in ArgoUML Variants

Gabriela Karoline Michelon<sup>1,2</sup>, Lukas Linsbauer<sup>1,3</sup>, Wesley K. G. Assunção<sup>4</sup>, Alexander Egyed<sup>1</sup>

<sup>1</sup>Institute for Software Systems Engineering - Johannes Kepler University Linz - Austria

<sup>2</sup>LIT Secure and Correct Systems Lab - Johannes Kepler University Linz - Austria

<sup>3</sup>Christian Doppler Laboratory MEVSS - Johannes Kepler University Linz - Austria

<sup>4</sup>COTSI - Federal University of Technology - Paraná, Toledo, Brazil

gabriela.michelon@jku.at,lukas.linsbauer@jku.at,wesleyk@utfpr.edu.br,alexander.egyed@jku.at

## ABSTRACT

Identifying and extracting parts of a system's implementation for reuse is an important task for re-engineering system variants into Software Product Lines (SPLs). An SPL is an approach that enables systematic reuse of existing assets across related product variants. The re-engineering process to adopt an SPL from a set of individual variants starts with the location of features and their implementation, to be extracted and migrated into an SPL and reused in new variants. Therefore, feature location is of fundamental importance to the success in the adoption of SPLs. Despite its importance, existing feature location techniques struggle with huge, complex, and numerous system artifacts. This is the scenario of ArgoUML-SPL, which stands out as the most used case study for the validation of feature location approaches. In this paper we use an automated feature location technique and apply it to the ArgoUML feature location challenge posed.

## CCS CONCEPTS

- Software and its engineering → Software product lines; Traceability; Software reverse engineering; Reusability.

## KEYWORDS

feature location, traceability, variants, clones, reuse, software product lines

### ACM Reference Format:

Gabriela Karoline Michelon<sup>1,2</sup>, Lukas Linsbauer<sup>1,3</sup>, Wesley K. G. Assunção<sup>4</sup>, Alexander Egyed<sup>1</sup>. 2019. Comparison-Based Feature Location in ArgoUML Variants. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/3336294.3342360>

## 1 INTRODUCTION

To conduct software maintenance or evolution, feature location is one of the key steps [17]. The goal of feature location is to establish mappings/traces between features and their respective implementation. Feature location allows practitioners to find and reason about the parts of a system that will be affected by modifications [6].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3342360>

Many feature location techniques can be found in literature [17]. In addition, an increasing interest and proliferation of techniques are observed in the field of re-engineering variants, developed using ad hoc reuse, into Software Product Lines (SPLs) [2, 3]. To evaluate these techniques, ArgoUML-SPL is one of the most used subject system, as shown in the ESPLA catalog [13].

Martinez et al. propose a challenge case study based on ArgoUML-SPL at the Systems and Software Product Line Conference (SPLC) [14]. This challenge defines the ArgoUML-SPL benchmark that is comprised of eight optional features and 15 predefined scenarios ranging from a single variant to 256 variants. The benchmark provides a set of tools and artifacts, including a ground truth to allow techniques to be evaluated and compared to each other. The main challenging characteristics of ArgoUML-SPL are: (i) it is a real software system of considerable size that resulted from a development process involving several developers; (ii) the implementation is composed of feature interactions and feature negations; (iii) the granularity of the implementation that must be traced varies from complete Java classes to statements inside methods.

Taking into account the challenges aforementioned, this work applies our feature location technique implemented in the ECCO tool [7, 11, 12] to the ArgoUML-SPL challenge [14] and presents and discusses the results. The contributions of this study are: (i) implementation of a Java adapter that extracts data from the Java source code files at the granularity asked for by the challenge (classes, methods and their respective refinements); (ii) application of ECCO to a real-world subject where we can show how well our approach deals with these system variants; (iii) providing and discussing results for the ArgoUML challenge case study so that others can compare their results to ours.

The remainder of this paper is structured as follows: Section 2 discusses related work. Section 3 briefly describes the challenge dataset. Section 4 describes our feature location technique and the methodology used to apply it to the posed challenge. Section 5 presents the results and offers a discussion with particularly interesting insights. Finally, Section 6 concludes.

## 2 RELATED WORK

Feature location is the first step in the process of re-engineering variants into SPLs [2]. To ease the migration of software variants into an SPL, some feature location techniques can automate the location of source code elements relevant to a given feature. ArgoUML-SPL is the subject system most used in this context [13].

Rubin and Chechik [17] describe the most common feature location techniques and categorize them by the strategies they use:

static program analysis, which leverages static dependencies between program elements; information retrieval techniques, based on information embedded in program identifier names and comments; and dynamic approaches that collect precise information about the program execution. In addition, hybrid approaches combine different techniques to take advantage of each approach.

Cruz et al. [5] provide a literature review and use ArgoUML-SPL to evaluate three information retrieval based feature location techniques. These strategies were compared based on their ability to correctly identify the source code of several features from the ArgoUML-SPL ground truth. The results suggest that Latent Semantic Indexing is better than both Paragraph Vectors and Latent Dirichlet Allocation. However, the values obtained for precision, recall and F1 measure were not satisfactory, and strategies to better deal with huge and complex artifacts are needed.

There are several comparison-based feature identification (e.g. [22]) or feature location (e.g. [1, 18, 21, 23]) techniques. Most of them rely on formal concept analysis (e.g. [1, 18, 21]). Many of them are hybrid approaches based on a combination of formal concept analysis and information retrieval techniques (e.g. [18, 21]) of which our approach uses neither. Most only consider single features and not their negations or interactions (i.e. conjunctions or disjunctions), can only be applied to specific types of implementation artifacts (e.g. source code), or only operate on a coarse level of implementation artifacts (e.g. class or method level and not statement level). Our approach has none of these restrictions.

Martinez et al. built the tool BUT4Reuse [15, 16]. BUT4Reuse supports extractive SPL adoption by providing a framework and techniques for feature identification, feature location, mining feature constraints, extraction of reusable assets, feature model synthesis and visualizations, etc. This tool is designed to be generic and extensible, allowing researchers to include their own strategies. To this end, they also employ an adapter concept in order to support variability in different types of implementation artifacts.

### 3 DATA SET

The ArgoUML-SPL challenge [14] provides 15 scenarios and a *ground truth* consisting of 24 traces. ArgoUML is an open source tool for UML modeling that is implemented in Java and was refactored into a product line [4]. It consists of two mandatory features: Diagrams Core, and Class Diagram; and eight optional features: State Diagram, Activity Diagram, Use Case Diagram, Collaboration Diagram, Deployment Diagram, Sequence Diagram, Cognitive Support, and Logging.

#### 3.1 Scenarios

The challenge provides 15 scenarios (see Table 1) with varying number of variants. In our context, a variant is defined as follows:

**DEFINITION.** A variant  $V$  is a pair  $(F, A)$  that maps a set of features  $F$  that the variant provides to a set of implementation artifacts  $A$  that implement the variant.

More specifically, *Original* scenario contains only the initial ArgoUML system as a single variant from which the ArgoUML-SPL was initially created [4]. *Traditional* scenario has 10 variants, one with all the features, one with only the mandatory features, and for every optional feature one variant with only that feature disabled.

**Table 1: ArgoUML Challenge Scenarios**

Scenario	Size	Description
Original	1	Original ArgoUML variant containing all features.
Traditional	10	Variants with no, all, and combinations of 7 optional features.
PairWise	9	Set of variants that covers all pairwise feature combinations.
2-10 Random	2-10	Randomly selected subsets of variants.
50 Random	50	Randomly selected subset of variants.
100 Random	100	Randomly selected subset of variants.
All	256	All possible variants of ArgoUML-SPL.

*PairWise* scenario is composed of 9 variants obtained using the pairwise feature coverage algorithm from FeatureIDE [20]. *Random* scenarios of different size (2, 3, 4, 5, 6, 7, 8, 9, 10, 50 and 100 variants) with randomly selected variants. *All* scenario has all the  $2^8 = 256$  possible variants of ArgoUML-SPL obtained from FeatureIDE [20].

#### 3.2 Ground Truth

The ground truth provided by the challenge consists of 24 traces.

**DEFINITION.** A trace  $T$  is a pair  $(F, A)$  that maps a propositional logic formula  $F$  whose literals are features to a set of implementation artifacts  $A$ .

In the context of this particular challenge, implementation artifacts represent Java code elements. In other words, every trace  $T$  maps a set of code elements  $T.A$  to a feature condition  $T.F$ .

The ground truth contains one trace for each of the eight individual features; Two traces with a single negative feature; 13 traces with a conjunction of two features; One trace with a conjunction of three features.

### 4 FEATURE LOCATION TECHNIQUE

We applied an automatic feature location technique that is based on the comparison of features and implementation artifacts of a set of variants [7, 10–12]. The approach was first presented at SPLC'13 [10] and has since evolved beyond the task of feature location (i.e. trace computation) to also support, for example, the composition of variants from the computed traces [9, 12]. It is now implemented in a publicly available tool called ECCO<sup>1,2</sup> [7, 12]. ECCO supports trace extraction (*commit* operation) and variant composition (*checkout* operation) with arbitrary types of implementation artifacts beyond just source code. To this end, it requires an adapter for every type of artifact that contains variability. To apply it to the posed challenge we developed a simple Java adapter to parse the code using JavaParser<sup>3,4</sup> [19].

Figure 1 shows the feature location process of ECCO. The entire implementation of a variant (in this case a set of Java files) is input to the responsible adapter (in this case the Java adapter). The adapter creates an artifact tree structure as illustrated in Figure 2. The *commit* operation uses the artifact tree structure and the set of features of the variant to compute traces from features to nodes in the artifact tree, which are finally stored in a repository. This process can be repeated incrementally with arbitrarily many variants. Every time a new variant is committed the traces in the repository

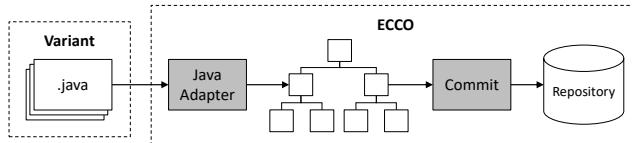
<sup>1</sup><https://jku-isse.github.io/ecco/>

<sup>2</sup><https://github.com/jku-isse/ecco>

<sup>3</sup><https://javaparser.org/>

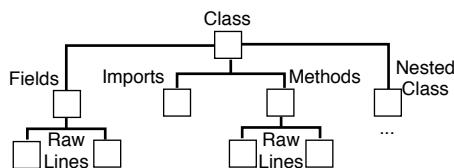
<sup>4</sup><https://github.com/javaparser>

are refined. One could, for example, decide to stop committing further variants when the traces in the repository have not changed anymore over the last few commits. Finally, the traces stored in the repository are exported into the format requested by the challenge and compared to the ground truth traces (see Section 3.2) to compute the challenge metrics (see Section 5.2).



**Figure 1: ECCO process for committing a single variant consisting of multiple Java files into its repository.**

Figure 2 illustrates the structure of the artifact tree created from Java files by the Java adapter. It reflects the requirements of the challenge that asks to trace classes, methods and their respective refinements in the form of changes to imports, fields and lines of code. Therefore, we store classes (including nested classes), imports, fields and methods explicitly. For fields and methods we also store their children (to be able to detect refinements) in the form of raw lines of code as they can span over multiple lines. This is necessary as variability in ArgoUML-SPL is often implemented at such a fine-granular level and annotations are not always placed in a disciplined manner [8]. Even single statements, e.g. field declarations, can span multiple lines of which only some might be annotated with features.



**Figure 2: Artifact tree structure created by the Java adapter.**

The *commit* operation is essentially based on five rules [11]. Assume two variants A and B:

- (1) Common artifacts (in A and B) *likely* trace to common features (in A and B).
- (2) Artifacts in A and not B *likely* trace to features in A and not B, and vice versa.
- (3) Artifacts in A and not B *do not* trace to features in B and not A, and vice versa.
- (4) Artifacts in A and not B *at most* trace to features in A, and vice versa.
- (5) Artifacts in A and B *at most* trace to features in A or B.

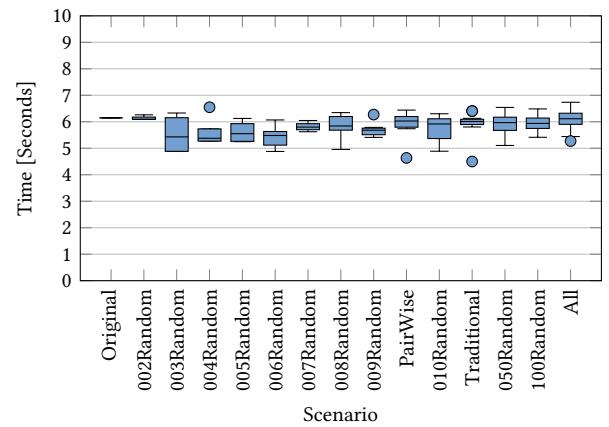
The first two rules quickly isolate features (or feature combinations) to which implementation artifacts likely trace. Rule 3 determines features to which implementation artifacts certainly cannot trace. The last two rules provide an upper bound on where features can at most trace. For more details please consult [11].

## 5 RESULTS

This section presents and discusses the results of our feature location technique ECCO being applied to each of the 15 scenarios. The results and instructions for reproducing them are publicly available<sup>5</sup>.

### 5.1 Time Performance

Figure 3 shows a box plot of the runtime per variant (i.e. *commit* operation) per scenario, ordered by increasing number of variants. It was measured on an HP ZBook 14 laptop, with Intel® Core™ i7-4600U processor (2.1GHz, 2 cores), 16GB of RAM and SSD storage, running Fedora Linux as operating system. Each scenario's average runtime per committed variant is between 4 and 7 seconds. Overall, the runtime remains quite constant. Fluctuations are most likely caused by differences in the size of variants.



**Figure 3: Runtime per Variant per Scenario**

### 5.2 Precision, Recall, F1 Score

To validate the computed traces, three metrics are computed for each trace  $T$ . They are calculated automatically by the challenge benchmark [14], which compares every ground truth trace  $T_{gt}$  with the respective (i.e. same feature condition  $T.F$ ) trace  $T_{ecco}$  computed by our feature location technique.

Precision is the percentage of correctly retrieved artifacts (i.e. code elements) relative to the total retrieved code elements.

$$\text{precision} = \frac{TP}{TP + FP} = \frac{|T_{gt}.A \cap T_{ecco}.A|}{|T_{ecco}.A|} \quad (1)$$

where TP (true positives) are the correctly retrieved code elements and FP (false positives) are the incorrectly retrieved code elements.

Recall is the percentage of correctly retrieved code elements relative to the total number of code elements in the ground truth.

$$\text{recall} = \frac{TP}{TP + FN} = \frac{|T_{gt}.A \cap T_{ecco}.A|}{|T_{gt}.A|} \quad (2)$$

where FN (false negatives) are code elements present in the ground truth which are not included in the retrieved code elements.

<sup>5</sup><https://github.com/jku-isse/SPLC2019-Challenge-ArgeUML-FeatureLocation>

The F1 score (F-measure) relates precision and recall and combines them into a single measure.

$$F1 = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} \quad (3)$$

Figure 4 shows average precision, recall and F1 score over all traces per scenario, ordered by increasing number of variants.

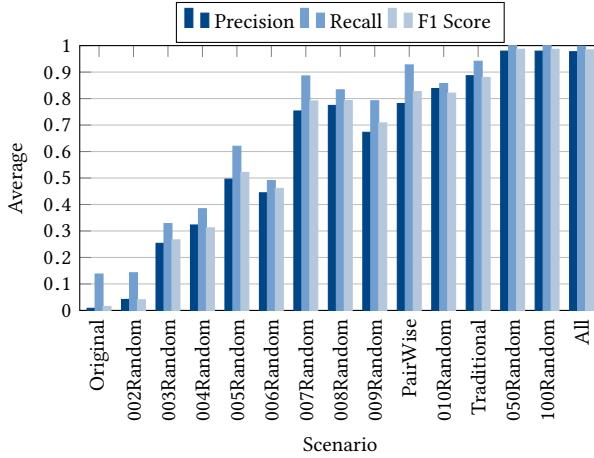


Figure 4: Average Precision, Recall and F1 Score per Scenario

The best results were obtained with the largest scenarios which achieved 100% average recall and around 99% average precision and F1 score. Overall, the results of our feature location technique improve the more variants are available. This is to be expected, as our technique is based on the comparison (i.e. commonalities and differences) of features and implementation of variants. As a consequence of this, scenarios consisting of only one or two variants produce quite useless results.

However, it is not generally true that more variants always produce better results. For example, the 9 random variants produce a slightly worse result than the 8 random variants. Similarly, the 10 traditional variants produce a slightly better result than the 10 random variants. This means that variants with *beneficial* configurations (i.e. combinations of features that exhibit *interesting* variability) can make up for a lower number of total variants available. Also, there seems to be a critical point after which the results do not improve much anymore, which is somewhere around 10 variants. After that, every additional variant only improves the results marginally. Beyond 50 variants the results do not even change at all anymore as nothing new can be learned from additional variants.

The few differences that remain in the results, even with all variants available for analysis, are caused by ambiguities during the alignment of sequences of lines of source code (for example, children of methods in the artifact tree, see Figure 2) during the comparison of the implementation of variants. Alignments of sequences of lines, i.e. insertions and deletions, do not always reflect perfectly the actual changes that were performed, as anyone who has ever used a source code diffing tool, e.g. when performing merges in a VCS such as Git, is probably aware. This causes misinterpretations of changes and leads to mismatches with the ground truth in some

cases. However, this does not mean that the computed traces are *wrong* as they still produce the exact same variants, it just means that there are multiple valid traces and that the one our approach computed does not match the one provided by the ground truth. This is illustrated with a minimalistic example in Figure 5. It shows three variants with features {A}, {A, B} and {B} respectively and alignments of the statement `i++` on the left and the corresponding traces that produce the variants in the form of annotated code on the right. The two rows illustrate two different alignments and corresponding different traces. Even though the traces (on the right) are different, the produced variants (on the left) are identical.

A	A, B	B	Traces
1 int i; 2 i++; 3 return i;	1 int i; 2 i++; 3 i++; 4 return i;	1 int i; 2 i++; 3 return i;	#if A i++; #end #if B i++; #end
1 int i; 2 i++; 3 return i;	1 int i; 2 i++; 3 i++; 4 return i;	1 int i; 2 i++; 3 return i;	#if A    B i++; #end #if A && B i++; #end

Figure 5: Illustration of two valid alignments of lines (top and bottom) in three variants with two optional features A and B (left side) and the corresponding traces (right side).

## 6 CONCLUSION

This work presents a solution to the ArgoUML-SPL feature location challenge posed at SPLC [14]. We applied ECCO, an automatic feature location technique, that is based on the comparison of features and implementation of a set of variants. Our technique computes a set of traces that map features (actually propositional logic formulas with features as literals) to code elements. Overall, the more variants are available the better our computed traces match the ground truth traces provided by the challenge. However, a critical point is reached somewhere around 10 variants where precision and recall reach around 90% and after that they only improve marginally with every additional variant. The highest gain per additional variant is achieved with the first few variants. The runtime of our technique increases linearly with the number of variants.

## ACKNOWLEDGMENTS

This research was funded by the JKU Linz Institute of Technology (LIT) and the state of Upper Austria, grant no. LIT-2016-2-SEE-019; the LIT Secure and Correct Systems Lab funded by the state of Upper Austria; the Austrian Science Fund (FWF), grant no. P31989; the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and KEBA AG, Austria; Pro2Future, a COMET K1-Centre of the Austrian Research Promotion Agency (FFG), grant no. 854184; and CNPq grant 408356/2018-9.

## REFERENCES

- [1] Ra'Fat Al-Msie'deen, Abdelhak Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. 2013. Feature Location in a Collection of Software Product Variants Using Formal Concept Analysis. In *Safe and Secure Software Reuse (ICSR)*, Vol. 7925. Springer, Berlin, Heidelberg, 302–307. [https://doi.org/10.1007/978-3-642-38977-1\\_22](https://doi.org/10.1007/978-3-642-38977-1_22)
- [2] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering* 22, 6 (Dec 2017), 2972–3016. <https://doi.org/10.1007/s10664-017-9499-z>
- [3] Wesley K. G. Assunção and Silvia R. Vergilio. 2014. Feature location for software product line migration: a mapping study. In *18th International Software Product Lines Conference - Companion Volume for Workshop, Tools and Demo papers (SPLC '14)*. ACM, Florence, Italy, 52–59. <https://doi.org/10.1145/2647908.2655967>
- [4] Marcus Vinicius Couto, Marco Túlio Valente, and Eduardo Figueiredo. 2011. Extracting Software Product Lines: A Case Study Using Conditional Compilation. In *15th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, Oldenburg, Germany, 191–200. <https://doi.org/10.1109/CSMR.2011.25>
- [5] Daniel Cruz, Eduardo Figueiredo, and Jabier Martinez. 2019. A Literature Review and Comparison of Three Feature Location Techniques using ArgoUML-SPL. In *13th International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 16.
- [6] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process* 25 (2013), 53–95. <https://doi.org/10.1002/smrv.567>
- [7] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *37th IEEE/ACM International Conference on Software Engineering*, Vol. 2. IEEE Computer Society, Florence, Italy, 665–668. <https://doi.org/10.1109/ICSE.2015.218>
- [8] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *10th International Conference on Aspect-Oriented Software Development, AOSD 2011, Porto de Galinhas, Brazil, March 21–25, 2011*, Paulo Borba and Shigeru Chiba (Eds.). ACM, 191–202. <https://doi.org/10.1145/1960275.1960299>
- [9] Lukas Linsbauer, Alexander Egyed, and Roberto Erick Lopez-Herrejon. 2016. A variability aware configuration management and revision control platform. In *38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016 - Companion Volume*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 803–806. <https://doi.org/10.1145/2889160.2889262>
- [10] Lukas Linsbauer, E. Roberto Lopez-Herrejon, and Alexander Egyed. 2013. Recovering Traceability Between Features and Code in Product Variants. In *17th International Software Product Line Conference (SPLC '13)*. ACM, New York, NY, USA, 131–140. <https://doi.org/10.1145/2491627.2491630>
- [11] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability extraction and modeling for product variants. *Software & Systems Modeling* 16, 4 (Oct 2017), 1179–1199. <https://doi.org/10.1007/s10270-015-0512-y>
- [12] Stefan Fischer Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *IEEE International Conference on Software Maintenance and Evolution*.
- [13] IEEE Computer Society, Victoria, BC, Canada, 391–400. <https://doi.org/10.1109/ICSME.2014.61>
- [14] Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. 2017. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *21st International Systems and Software Product Line Conference - Volume B (SPLC '17)*. ACM, New York, NY, USA, 38–41. <https://doi.org/10.1145/3109729.3109748>
- [15] Jabier Martinez, Nicolas Ordonez, Xhevahire Ternava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Túlio Valente. 2018. Feature Location Benchmark with ArgoUML SPL. In *22Nd International Systems and Software Product Line Conference - Volume 1 (SPLC '18)*. ACM, New York, NY, USA, 257–263. <https://doi.org/10.1145/3233027.3236402>
- [16] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Bottom-up adoption of software product lines: a generic and extensible approach. In *19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20–24, 2015*, Douglas C. Schmidt (Ed.). ACM, 101–110. <https://doi.org/10.1145/2791060.2791086>
- [17] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2017. Bottom-up technologies for reuse: automated extractive adoption of software product lines. In *39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 67–70. <https://doi.org/10.1109/ICSE-C.2017.15>
- [18] Julie Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering, Product Lines, Languages, and Conceptual Models*. Springer, Berlin, Heidelberg, 1–51. [https://doi.org/10.1007/978-3-642-36654-3\\_2](https://doi.org/10.1007/978-3-642-36654-3_2)
- [19] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. 2018. *JavaParser for Processing Java Code*. <https://javaparser.org/>
- [20] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85. <https://doi.org/10.1016/j.scico.2012.06.002>
- [21] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. 2012. Feature Location in a Collection of Product Variants. In *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15–18, 2012*. IEEE Computer Society, 145–154. <https://doi.org/10.1109/WCRE.2012.24>
- [22] Tewfik Ziadi, Luz Frias, Marcos Aurélio Almeida da Silva, and Mikal Ziane. 2012. Feature Identification from the Source Code of Product Variants. In *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27–30, 2012*, Tom Mens, Anthony Cleve, and Rudolf Ferenc (Eds.). IEEE Computer Society, 417–422. <https://doi.org/10.1109/CSMR.2012.52>
- [23] Tewfik Ziadi, Christopher Henard, Mike Papadakis, Mikal Ziane, and Yves Le Traon. 2014. Towards a language-independent approach for reverse-engineering of software product lines. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 – 28, 2014*, Yookun Cho, Sung Y. Shin, Sang-Wook Kim, Chih-Cheng Hung, and Jiman Hong (Eds.). ACM, 1064–1071. <https://doi.org/10.1145/2554850.2554874>

# Migrating Java-Based Apo-Games into a Composition-Based Software Product Line

Jamel Debbiche

Chalmers | University of Gothenburg  
Gothenburg, Sweden

Jacob Krüger

Otto-von-Guericke University  
Magdeburg, Germany

## ABSTRACT

A software product line enables an organization to systematically reuse software features that allow to derive customized variants from a common platform, promising reduced development and maintenance costs. In practice, however, most organizations start to clone existing systems and only extract a software product line from such clones when the maintenance and coordination costs increase. Despite the importance of extractive software-product-line adoption, we still have only limited knowledge on what practices work best and miss datasets for evaluating automated techniques. To improve this situation, we performed an extractive adoption of the Apo-Games, resulting in a systematic analysis of five Java games and the migration of three games into a composition-based software product line. In this paper, we report our analysis and migration process, discuss our lessons learned, and contribute a feature model as well as the implementation of the extracted software product line. Overall, the results help to gain a better understanding of problems that can appear during such migrations, indicating research opportunities and hints for practitioners. Moreover, our artifacts can serve as dataset to test automated techniques and developers may improve or extend them in the future.

## CCS CONCEPTS

- Software and its engineering → Software product lines; Software reverse engineering; Maintaining software.

## KEYWORDS

Software product line, Extraction, Case Study, Feature model, FeatureHouse, Apo-Games

### ACM Reference Format:

Jamel Debbiche, Oskar Lignell, Jacob Krüger, and Thorsten Berger. 2019. Migrating Java-Based Apo-Games into a Composition-Based Software Product Line. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/3336294.3342361>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-7138-4/19/09...\$15.00  
<https://doi.org/10.1145/3336294.3342361>

Oskar Lignell

Chalmers | University of Gothenburg  
Gothenburg, Sweden

Thorsten Berger

Chalmers | University of Gothenburg  
Gothenburg, Sweden

## 1 INTRODUCTION

A *software product line* allows an organization to implement a family of similar software products based on a common platform [1, 9]. In such a platform, developers systematically manage and reuse *features* that implement mandatory and optional functionalities of the products. Developers can derive a specific product by configuring (selecting) the features that shall be part of the product, which is then built in an automated step. Software product lines promise several benefits, such as reduced development and maintenance costs, improved quality, and faster time to market [14, 34]. Still, despite such benefits, most organizations fear the initially higher investment to set up a reusable platform [8, 20] and only later *extract* [15] one out of a set of existing software variants. In practice, such variants often emerge from cloning one product and adapting it to the needs of another customer, referred to as *clone-and-own* [7, 11, 32].

While the extractive approach of software-product-line adoption is the most common one in practice [6, 12], most techniques (e.g., feature location) that aim to automatically analyze the cloned systems face severe limitations [18, 22, 23, 30]. To overcome such limitations, we need to provide common ground truths that allow to evaluate and compare techniques based on real-world artifacts [22, 33]. As a step in this direction, Krüger et al. [21] have contributed a challenge case that comprises 20 Java and five Android games that a single developer implemented based on the *clone-and-own* approach. They have challenged the research community to provide additional artifacts (i.e., feature models, feature locations, code smells, architectures, migrated software product lines) that can serve as datasets to evaluate automated techniques for analyzing these games.

In this paper, we report our experiences on tackling the fifth challenge of migrating game variants into a software product line. To this end, the first two authors of this paper analyzed five Java games and migrated three of these (due to time restrictions) into a composition-based software product line implemented with *FeatureHouse* [2]. We further documented our activities, results, and experiences for each step. Our contributions are:

- We describe the applied analysis and migration process that resulted in the extracted software product line.
- We report the challenges that we experienced during this process as lessons learned.
- We provide a repository<sup>1</sup> with all artifacts we created, namely the feature model and code base for the software product line.

<sup>1</sup>[https://bitbucket.org/Jacob\\_Krueger/splc2019\\_featurehouse\\_apo-games\\_spl](https://bitbucket.org/Jacob_Krueger/splc2019_featurehouse_apo-games_spl)

**Table 1: Selected subject systems for this study. Asterisks (\*) denote games that we analyzed, but did not transform.**

Name	Year	SLOC	Game Type
* ApoCheating	2006	3,960	Level-based puzzles
* ApoStarz	2008	6,454	Level-based puzzles
Apolcarus	2011	5,851	Endless runner
ApoNotSoSimple	2011	7,558	Level-based puzzles
ApoSnake	2012	6,557	Level-based puzzles

For simplicity, the repository comprises a FeatureIDE [26] project that others can import.

Our results are helpful for researchers and practitioners to better understand how variants can be migrated into a software product line. Moreover, researchers can use our implementation as baseline to evaluate and compare automated techniques or to incorporate more Apo-Games in the future.

## 2 METHODOLOGY

As we tackle the fifth challenge defined by Krüger et al. [21], we migrated cloned variants towards a software product line. Consequently, we also performed parts of other challenges (e.g., feature modeling) and provide corresponding artifacts (i.e., the feature model). In this section, we describe our applied methodology, comprising the *subject systems* we selected, our *preparation* for the transformation, the *feature recovery* process, and our actual *transformation* process. We remark that we did not follow a specific methodology (e.g., as described by Assunção et al. [3]), but employed and adapted activities that are regularly mentioned in the literature. Finally, we summarize this section by describing the *resulting software product line*.

### 2.1 Subject Systems

The Apo-Games comprise a set of 20 Java games that have been developed with the clone-and-own approach. Of these games, we selected five as subjects and provide an overview of these five in Table 1. We remark that we selected games that showed the most commonalities while playing them, indicating that they would contribute to a reasonable software product line. As we can see, the games are from different periods and cover between four to over seven thousand source lines of code. While four of them are part of the same sub-domain of games (puzzles), one is a representative for an endless running game. We included this one game from a different sub-domain, because we assumed more overlap for the same type of games, but also aimed to show that we can integrate rather different games into a software product line.

During our analysis, we faced some problems (cf. Section 3), due to which we only transformed three of the games. Moreover, we cleaned the source code to remove dead code with the Eclipse plug-in UCDetector<sup>2</sup> in several variants, reducing the code base by almost 40% (removing 11,670 out of 30,380 SLOC). Most of the dead code resulted from the developer creating a clone without removing unused code afterwards, for example, for enemy entities. Finally, we translated the German comments in the games into English to support our program comprehension. This was done using Google

<sup>2</sup><https://marketplace.eclipse.org/content/unnecessary-code-detector>

Translate, however, most of the comments were not helpful as it was tacit knowledge, such as describing setters and getters.

### 2.2 Preparation

Before the actual transformation, we analyzed existing tools to identify whether we could rely on one of them. However, we found that most existing techniques (e.g., for automatic feature location [30]) depend on specific artifacts (e.g., source code, models, documentation, git history) or additional tools that must be available to the developer. A particular problem in this regard is that most tools were not useful in our case, as they are discontinued, commercial or provide unsuitable results. For instance, But4Reuse [25] can suggest features and support extractive adoption, but the suggested features did not align to the domain features we identified in our top-down analysis. Namely, But4Reuse suggested features, such as *mousebuttonfunction*, according to keywords that appear often in the code, but these keywords do not reveal the actual domain features. Due to this mismatch, we decided to rely on a manual transformation, resulting in a feature-wise migration of the games.

While we did not find a technique that we could employ as is for the Apo-Games, we nonetheless adopted the described strategies for our own process. This led to the adoption of the sandwich approach [35] for our feature recovery process (cf. Section 2.3). We used the results, especially the feature mappings, to plan the transformation and to actually migrate variant features. During our analysis, we also decided to use FeatureIDE [26], as it supports various activities and variability mechanisms that we needed. Moreover, FeatureIDE is a plug-in for Eclipse, which allowed us to use other plug-ins more easily. For the variability mechanism, we used a composition-based technique and feature-oriented programming [29] in particular, supporting the physical separation of features [1, 13, 16]. Consequently, we selected FeatureHouse [2], which is directly supported by FeatureIDE and integrates more recent Java versions (i.e., compared to AHEAD [5]).

### 2.3 Feature Recovery

The first step of extracting a software-product line is to detect features and their dependencies in the legacy systems [4], which we defined in a feature model [1, 10]. Moreover, we had to also locate and map features in the source code [18]. To this end, we decided to employ a manual analysis comprising *top-down* and *bottom-up* strategies, referred to as sandwich approach [35].

**Top-Down Analysis.** As first step of our top-down analysis, we played each subject game and identified its visible features. We listed the features and performed a pairwise comparison between the games to identify commonalities and variability. In a second step, we reverse engineered class diagrams for each game, using the Visual Paradigm<sup>3</sup> plug-in for Eclipse. This extraction helped to understand that, except for ApoCheating, all subject games share the same architecture.

During the top-down analysis, we identified 33 features throughout all games (i.e., the ones we could not match to But4Reuse's suggestions). However, when we started to investigate the actual source code to map these features, we found that several technical features were still missing in our documentation. To address

<sup>3</sup><https://marketplace.eclipse.org/content/visual-paradigm-eclipse>

**Table 2: Statistics of the extracted software product line compared to the original games. Products refers to the games that we could execute based on the implemented features.**

Feature Model Statistics				
Features	Concr.	Impl.	Constraints	Products
47	42	23	16	56
Source Code Statistics		Legacy Statistics		
Classes	SLOC	SLOC	Legacy	Reduction
55	13,932		19,966	-30.22%

Concr.: Concrete features; Impl.: Implemented features

this problem, we performed a thorough code review to identify features that were not apparent from the user interfaces or only partly implemented.

**Bottom-Up Analysis.** For our code review, we again employed a pairwise comparison of the games. To this end, we started on project level (folder and class names) and continued with the actual source code. During this analysis, we relied on Code Compare.<sup>4</sup> As this tool only flags whether (1) two files have identical names and contents, (2) vary in content or (3) are completely unique, we further compared files manually in Eclipse.

Besides identifying 14 more features, our manual analysis also allowed us to locate and map the source code that belongs to each feature. We documented the mapping in a separate table, collecting the methods and classes that contribute to a feature. This table was also useful to derive information about the features, such as their tangling and scattering.

**Feature Modeling.** After our analysis and mapping, we finalized a feature model that could represent the existing variants and their 47 features. We show an excerpt of the feature model in Figure 1, which we describe in more detail in Section 2.5. As feature models can have various identical representations, it was quite challenging to evaluate what design would be best for our work. For instance, Mendonça et al. [27] show specifically for the Apo-Games that various feature models are Pareto-equivalent depending on the objectives (e.g., representing only the legacy games). We decided to add as much variability as possible, instead of having a feature model that can solely represent the legacy games—which seems more reasonable to facilitate the evolution of a software product line, during which corresponding configuration options would also be added.

## 2.4 Transformation

As first step of the actual transformation, we migrated the common code base of all games into a FeatureIDE project. This common part was rather small, comprising the main panel and the game engine. While this base could already be compiled for testing, it only showed a black window.

We then incrementally added features of the games into the software product line. A particular problem in this regard was to ensure the correct behavior of the transformed code, as we needed at least one complete game for testing. Moreover, we had to carefully plan which features to implement first, due to their domain and technical dependencies. We partly addressed the dependencies by changing different data structures into more flexible and variable

ones. For instance, we changed most arrays into data structures of the List collection to ensure that the games were executable despite missing or disabled features. However, such changes and the remaining dependencies still challenged testing.

During the transformation, we also added variant-specific features that do not contribute to reuse, but only variability. Due to time restrictions, we finally migrated 23 features, which can be used to instantiate three of the legacy games (cf. Table 1), and for which we show their dependencies in Figure 1. Overall, we emphasized variability over reuse, and thus introduced not only features based on clones and variations, but actual domain features [23]. As a result, the software product line allows to configure 56 games.

## 2.5 The Software Product Line

In Table 2, we provide a brief overview of the software product line we extracted. As aforementioned, we identified and modeled 47 features with 16 cross tree constraints. We show the 23 implemented features and their dependencies as a feature model in Figure 1. In the diagram, we can see that only few abstract features exist for structuring other features. On the positive side, we have only few features that are directly connected to a specific game or game type (e.g., *SnakeInteractive*). Thus, most of the features we identified and implemented seem well suited for reuse.

Considering the source code, we implemented 55 classes comprising 13,932 SLOC in our software product line. As the three legacy games totaled at 19,966 SLOC, we achieved a reduction of roughly 30%. Some of this reduction is the result of our code cleansing in the beginning. However, this rather shows the positive impact an extractive adoption can have on code quality. Also, composition-based variability mechanisms usually result in additional source code, because new classes are needed to implement feature modules. So, we argue that our software product line resulted in a reasonable complexity and size on implementation level.

Finally, our software product line allows to configure 56 games, not only the three legacy games, showing that a variable platform can immediately increase the product portfolio. To test the correct behavior, we configured, generated, and ran all 56 games successfully. However, we remark that not all configurations result in a fully playable game, yet. This is especially true for games that contain ApoIcarus game-play, as this variant is not completely migrated nor as configurable as the other two games (ApoNotSoSimple and ApoSnake). Due to the changes we employed during the transformation, a code-level comparison of the original games and the instantiated variants is not useful, but we were able to play the original games as variants of the software product line.

## 3 LESSONS LEARNED

During the transformation of the three games into a software product line, we experienced five main challenges.

**Abstraction Level of Features.** While identifying features with the sandwich approach, we found that different abstraction levels can result in varying sets of features. This highlights the importance of combining such analyses. Moreover, this experience underpins that we need to decide on the purpose and extent of a feature model before its actual design, also deciding how to structure features [28]. Despite the mismatch that we experienced with But4Reuse, we also

<sup>4</sup><https://www.devart.com/codecompare/>

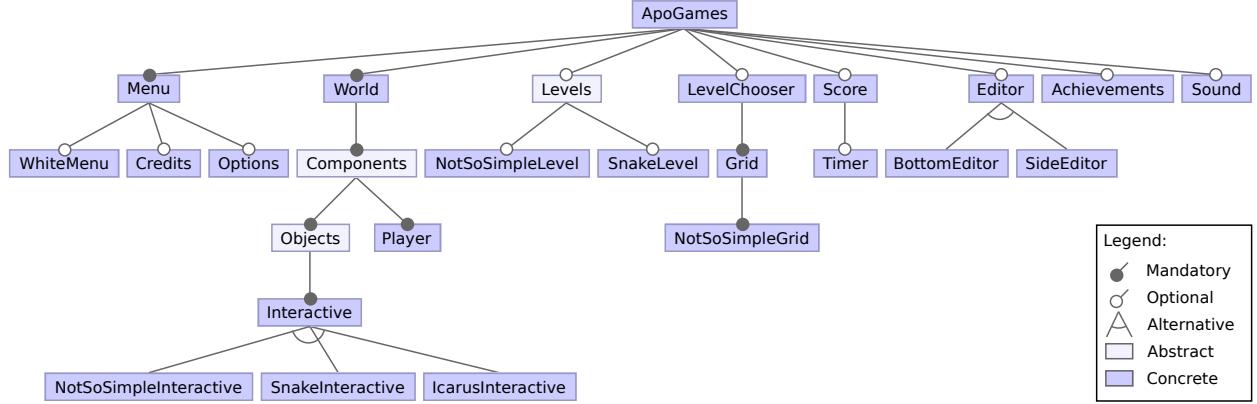


Figure 1: Feature model of the software product line showing implemented features only and no cross-tree constraints.

argue that such tools can support the semi-automated identification of features on the technical level.

**Planning of Features.** For the migration of a software product line, limited resources are available. We experienced that careful planning and extending analysis activities can considerably simplify the actual transformation of the variants. Most importantly in our experience was to know what dependencies between features and to their artifacts exist. This knowledge helped to get an intuition of the effort that would be necessary during the actual migration.

**Updates on the Feature Model.** While we implemented features, we experienced that we identified and added new constraints to the feature model. However, this was problematic as, at the beginning, many features were not fully functioning, meaning that many constraints were missing, too. Thus, constant updates to the feature model and the corresponding source code added effort to the extraction process.

**Complexity of Superimposition.** FeatureHouse uses superimposition to compose various features with the same class and method names, resulting in a customized variant. While this works properly and is a concept related to inheritance, it also poses challenges when the size of a project grows. As different features can have classes with the same name and physically separate related code, it became challenging to understand and identify what classes to change during maintenance and updates [17, 19, 31].

**Tracking Evolution.** We used a version control system and different branches to document our progress. Still, we found it challenging to understand later on what happened at what point in time, as feature code was scattered and tangled throughout commits—which is a good argument to use *variation control systems* [24]. For instance, two features may be changed to enable a third one (e.g., implementing updates to the feature model) and all changes are part of one commit.

#### 4 THREATS TO VALIDITY

**Internal Validity.** The major threat to the results of this work is the missing interaction with the original developer. Instead, we relied on code analysis, reading comments, and reverse engineering architectures. Consequently, while we carefully analyzed and checked the results, we cannot ensure that we understood all parts of the Apo-Games perfectly or that another (i.e., the original) developer would derive the same implementation for a software product

line. Still, as we were able to instantiate and run the original and 53 more games, we argue that our implementation is reasonable and can serve as a dataset for transformation and analysis techniques or to incrementally add more games.

**External Validity.** The subject systems are small compared to industrial or established open-source systems. However, they are publicly available and have been truly developed based on the clone-and-own approach, for which only few real-world subject systems exist. In addition, games contribute to more and more software systems, also exhibiting similar development patterns and characteristics as other software [7]. Thus, we cannot overly generalize the results, but they still yield important insights into the extraction of software product lines.

## 5 CONCLUSION

In this paper, we described a case study during which we migrated three Java-based Apo-Games into a composition-based software product line. For this purpose, we conducted a manual analysis and transformation process, resulting in the following insights:

- Extracting a composition-based software product line is challenging and time consuming, due to the changes that are needed to enable composition.
- During code transformations, we highly recommend to ensure that the software product line can always be tested to ensure the correct transformation of features.
- Incrementally adopting new features facilitates the extraction, as various artifacts (i.e., the feature model) need updates and must be tested.

Besides our lessons learned, we also provide a public repository comprising the extracted software product line and feature model.

In future work, we aim to extend the current artifacts and provide more detailed insights into the migration process. To this end, we want to replicate the extraction to verify our results and improve the validity of our insights. Finally, we plan to improve our dataset so that we can use it as ground-truth to evaluate automated techniques for extracting software product lines.

**Acknowledgments.** This work is supported by the ITEA project REVaMP<sup>2</sup> funded by Vinnova Sweden (2016-02804), and the Swedish Research Council Vetenskapsrådet (257822902). We thank Jennifer Horkoff for valuable comments on this work.

## REFERENCES

- [1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.
- [2] Sven Apel, Christian Kästner, and Christian Lengauer. 2011. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering* 39, 1 (2011), 63–79.
- [3] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Reengineering Legacy Applications into Software Product Lines: A Systematic Mapping. *Empirical Software Engineering* 22, 6 (2017), 2972–3016.
- [4] Wesley K. G. Assunção and Silvia R. Vergilio. 2014. Feature Location for Software Product Line Migration: A Mapping Study. In *International Software Product Line Conference (SPLC)*. ACM, 52–59.
- [5] Don Batory. 2004. Feature-Oriented Programming and the AHEAD Tool Suite. In *International Conference on Software Engineering (ICSE)*. IEEE, 702–703.
- [6] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 7:1–7:8.
- [7] John Businge, Openja Moses, Sarah Nadi, Engineer Bainomugisha, and Thorsten Berger. 2018. Clone-Based Variability Management in the Android Ecosystem. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 625–634.
- [8] Paul C. Clements and Charles W. Krueger. 2002. Point / Counterpoint: Being Proactive Pays Off / Eliminating the Adoption Barrier. *IEEE Software* 19, 4 (2002), 28–31.
- [9] Paul C. Clements and Linda M. Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley.
- [10] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)*. ACM, 173–182.
- [11] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34.
- [12] Slawomir Duszynski, Jena Knodel, and Martin Becker. 2011. Analyzing the Source Code of Multiple Software Variants for Reuse Potential. In *Working Conference on Reverse Engineering (WCRE)*. IEEE, 303–307.
- [13] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2009. A Model of Refactoring Physically and Virtually Separated Features. In *International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 157–166.
- [14] Peter Knauber, Jesus Bermejo, Günter Böckle, Julio C. S. do Prado Leite, Frank J. van der Linden, Linda M. Northrop, Michael Stark, and David M. Weiss. 2002. Quantifying Product Line Benefits. In *International Workshop on Software Product-Family Engineering (PFE)*. Springer, 155–163.
- [15] Charles W. Krueger. 2002. Easing the Transition to Software Mass Customization. In *International Workshop on Software Product-Family Engineering (PFE)*. Springer, 282–293.
- [16] Jacob Krüger. 2017. Lost in Source Code: Physically Separating Features in Legacy Systems. In *International Conference on Software Engineering (ICSE)*. IEEE, 461–462.
- [17] Jacob Krüger. 2018. Separation of Concerns: Experiences of the Crowd. In *Symposium on Applied Computing (SAC)*. ACM, 2076–2077.
- [18] Jacob Krüger, Thorsten Berger, and Thomas Leich. 2019. *Software Engineering for Variability Intensive Systems*. CRC Press, Chapter Features and How to Find Them: A Survey of Manual Feature Location, 153–172.
- [19] Jacob Krüger, Güll Calikli, Thorsten Berger, Thomas Leich, and Gunter Saake. 2019. Effects of Explicit Feature Traceability on Program Comprehension. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM. Accepted.
- [20] Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. 2016. Extracting Software Product Lines: A Cost Estimation Perspective. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 354–361.
- [21] Jacob Krüger, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. 2018. Apo-Games - A Case Study for Reverse Engineering Variability from Cloned Java Variants. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 251–256.
- [22] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is my Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software* 152 (2019), 239–253.
- [23] Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2017. Finding Lost Features in Cloned Systems. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 65–72.
- [24] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 49–62.
- [25] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Bottom-up Adoption of Software Product Lines: A Generic and Extensible Approach. In *International Conference on Software Product Line (SPLC)*. ACM, 101–110.
- [26] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.
- [27] Willian D. F. Mendonça, Wesley K. G. Assunção, and Lukas Linsbauer. 2018. Multi-Objective Optimization for Reverse Engineering of Apo-Games Feature Models. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 279–283.
- [28] Damir Nešić, Jacob Krüger, Stefan Stănciulescu, and Thorsten Berger. 2019. Principles of Feature Modeling. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM. Accepted.
- [29] Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 419–443.
- [30] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering*. Springer, 29–58.
- [31] Janet Siegmund, Christian Kästner, Jörg Liebig, and Sven Apel. 2012. Comparing Program Comprehension of Physically and Virtually Separated Concerns. In *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 17–24.
- [32] Stefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 151–160.
- [33] Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *International Systems and Software Product Line Conference (SPLC)*. ACM. Accepted.
- [34] Frank van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action*. Springer.
- [35] Yinxing Xue. 2011. Reengineering Legacy Software Products into Software Product Line Based on Automatic Variability Analysis. In *International Conference on Software Engineering (ICSE)*. ACM, 1114–1117.

# Migrating the Android Apo-Games into an Annotation-Based Software Product Line

Jonas Åkesson

Chalmers University of Technology  
Gothenburg, Sweden

Jacob Krüger

Otto-von-Guericke University  
Magdeburg, Germany

## ABSTRACT

Most organizations start to reuse software by cloning complete systems and adapting them to new customer requirements. However, with an increasing number of cloned systems, the problems of this approach become severe, due to synchronization efforts. In such cases, organizations often decide to extract a software product line, which promises to reduce development and maintenance costs. While this scenario is common in practice, the research community is still missing knowledge about best practices and needs datasets to evaluate supportive techniques. In this paper, we report our experiences with extracting a preprocessor-based software product line from five cloned Android games of the Apo-Games challenge. Besides the process we employed, we also discuss lessons learned and contribute corresponding artifacts, namely a feature model and source code. The insights into the processes help researchers and practitioners to improve their understanding of extractive software-product-line adoption. Our artifacts can serve as a valuable dataset for evaluations and can be extended in the future to support researchers as a real-world baseline.

## CCS CONCEPTS

- Software and its engineering → Software product lines; Software reverse engineering; Maintaining software.

## KEYWORDS

Software product line, Extraction, Case study, Feature model, Antenna, Apo-Games

### ACM Reference Format:

Jonas Åkesson, Sebastian Nilsson, Jacob Krüger, and Thorsten Berger. 2019. Migrating the Android Apo-Games into an Annotation-Based Software Product Line. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/3336294.3342362>

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-7138-4/19/09...\$15.00  
<https://doi.org/10.1145/3336294.3342362>

Sebastian Nilsson

Chalmers University of Technology  
Gothenburg, Sweden

Thorsten Berger

Chalmers | University of Gothenburg  
Gothenburg, Sweden

## 1 INTRODUCTION

In practice, organizations often start to develop a family of similar software systems by copying an existing one and adapting it to new requirements. This approach is called clone-and-own [6, 21], operating at the granularity of whole systems—or, in our case, Android apps [4]. While it is a simple and quick approach to software reuse, it also has severe limitations. All changes, such as bug fixes and updates, must be propagated to ensure the correct behavior of all clones, each of which can have individual side effects. Thus, maintaining a large number of clones becomes an error-prone and costly activity [6, 18, 24].

When these problems become too severe, organizations often start to extract [9] a *software product line* from the cloned legacy systems [1, 5]. A software product line promises benefits considering the quality, development effort, and time-to-market, mainly through the reusable features that are implemented in a common platform [8, 23]. Instead of having separated code clones for each system, developers can configure features (i.e., select or deselect them) to define a variant that is automatically created.

Despite the practical importance of migrating from cloned systems into a software product line [3, 7, 24], we still lack detailed insights into best practices, and most automated techniques are of limited applicability [10, 11, 13, 14, 19, 22]. To address these shortcomings, more and more researchers propose to systematically collect real-world case studies and datasets, for instance, in the EPSLA catalog [16]. Besides collecting experiences of researchers and practitioners, openly accessible data does also provide ground-truths to evaluate and benchmark automated techniques [13, 22]. In this spirit, Krüger et al. [12] provide a set of 25 games (20 Java, five Android) that have been implemented by a single developer using the clone-and-own approach. The authors asked the research community to work on five challenges to provide artifacts that can later be used for evaluation purposes. These challenges are concerned with reverse engineering (i.e., feature models, feature locations, architecture recovery), code analysis (i.e., code smells), and extracting an actual software product line.

In this challenge solution, we describe a case study that we conducted to address the last challenge. For this purpose, the first two authors of this paper extracted a software product line from the Android versions of the Apo-Games. As variability mechanism for our platform, we used the Antenna preprocessor. In this paper, we report our analysis and migration methodology, as well as experiences we gained during this process. More precisely, we make the following contributions in this paper:

**Table 1: Systems of the Apo-Games that we migrated.**

Name	Year	SLOC	Game Type
ApoClock	2012	3,615	Arcade/puzzles
ApoDice	2012	2,523	Level-based puzzles
ApoSnake	2012	2,965	Snake
ApoMono	2013	6,487	Level-based puzzles
MyTreasure	2013	5,360	Level-based puzzles

- We explain how we analyzed and migrated five Android games into an annotation-based software product line.
- We discuss our experiences on problems and open challenges that we faced during this process.
- We contribute a repository,<sup>1</sup> including the feature model and source code of our software product line. To this end, we provide a FeatureIDE [17] project that allows to configure and instantiate variants.

The results provide insights for researchers and practitioners alike to understand migration processes from cloned systems to a software product line. In addition, our artifacts can be used as baselines to evaluate and compare automated techniques. To this end, they can also serve as starting points to extend the artifacts further and integrate them into suitable datasets.

## 2 METHODOLOGY

Before the conduct of our case study, the first two authors of this paper performed a literature survey to familiarize with software-product-line techniques and tools. We remark that we did not employ a specific re-engineering process, such as those described by Assunção et al. [2], but employed commonly mentioned activities. To migrate the cloned Android games into a software product line, we also tackled additional challenges, mainly constructing a feature model. Within this section, we describe our methodology, which includes reports on our *subject systems*, the *domain analysis*, *feature recovery*, and the actual *transformation*.

### 2.1 Subject Systems

The Apo-Games case [12] includes a set of five different Android games for which the source code is publicly available. We provide an overview of these games in Table 1. As we can see, these five games have been published within two years and comprise between roughly 2,000 and 6,500 source lines of code. Each of these games uses a customized version of a third-party game engine.

### 2.2 Domain Analysis

The first step of our case study was to perform a domain analysis. We started by installing all games from Google Play and documented the games' functionalities and visible entities (e.g., buttons, player character, game logic). Based on this documentation, we gained knowledge about the commonalities and variations within the games, which we can characterize as follows:

- **ApoClock** provides two different game modes, arcade and puzzle. In both modes, the player has to hit clocks with a ball before any of the clocks runs out of time. While the arcade

<sup>1</sup>[https://bitbucket.org/Jacob\\_Krueger/splc2019\\_antenna\\_apo-games\\_spl](https://bitbucket.org/Jacob_Krueger/splc2019_antenna_apo-games_spl)

**Table 2: Legacy compared to software-product-line games.**

	#F	#A	#C	# Files	SLOC
Legacy	—	—	—	117	20,950
SPL	61	19	14	43	10,278
Reduction			-63.25%		-50.94%

F: Features; A: Abstract Features; C: Constraints

mode is endless and accumulates a score, the goal in the puzzle mode is to clear a designed level.

- **ApoDice** is a puzzle game that shows dices to the player, each dice having a number indicating how often it can move. The player has to move each dice to a black square on the board within the given number of moves. A total of 30 levels is predefined, but users can add their own ones.
- **ApoMono** requires the player to move an avatar from a starting position to a goal. Throughout the avatar's path, there are different obstacles, for example, missing tiles or walls. The player can move some stones on the field to overcome such obstacles.
- **ApoSnake** is similar to the original snake game where the player controls a snake to collect candy, increasing its size. However, in ApoSnake, the amount of candies in a level is fixed and they have different colors, which colors the snake in the same way. The snake can then move through any wall that has the same color as itself.
- **MyTreasure** is a puzzle game within a two dimensional maze. The player has to collect a golden coin in this maze, for which they has to rotate the maze, allowing the avatar to move according to gravity. In addition, yellow blocks in the maze also move according to the rotation.

All of these games also have an editor that allows the user to create own custom levels for that specific game. Custom levels are stored together with all levels other users created, and all levels can be retrieved and played by using a button in the menu. However, we found that this button crashes some games (except ApoMono and MyTreasure), because the server on which the games were stored is not available anymore.

In addition to playing the games, we performed a lightweight architecture analysis. For this purpose, we reverse engineered the class diagrams of all games using IntelliJ IDEA Ultimate. We manually inspected the classes that were shown in the models and compared the code and models. This helped us to identify feature locations and understand that all games share a common core and comprise some unique classes, for example, for parts of the game logic or branding. Moreover, we found that the systems were not solely cloned, but already designed for reuse, for instance, there are common classes to define the basic functions of all buttons.

Finally, we built the source code using Android Studio<sup>2</sup> to make sure that the games build and run as expected. We found that ApoSnake did not start, due to an issue in the external games engine. Thus, while we migrated its features, we were not able to start it, neither as legacy nor as software-product-line game.

<sup>2</sup><https://developer.android.com/studio/index.html>

### 2.3 Feature Recovery

Our domain analysis helped us to obtain a general understanding of the games and their features. To improve this understanding, we analyzed the source code of the games based on clone detection and pairwise comparison, an idea proposed by other researchers [7, 14]. **Clone Detection.** For code clone detection, we used the CPD tool.<sup>3</sup> We then analyzed the identified clones on file level, using the pairwise comparison of Meld.<sup>4</sup> To allow Meld to work properly, we first had to address the naming conventions of the Apo-Games [14], removing the game-specific prefixes. We considered identical code to be part of common features, while we considered variations for variable child features. Such a pair-wise comparison does not scale for many systems, but is feasible for comparing five games [7, 14]. **Feature Modeling.** We used FeatureIDE [17] to model the identified commonalities and variability in a feature model. To derive the dependencies between features, we relied on our domain knowledge and the presence or absence of features in the legacy games (e.g., there are game-specific features that have to be in alternative groups). In Figure 1, we display the final feature model that we constructed after the previous steps.

### 2.4 Transformation

We used the preprocessor Antenna from FeatureIDE to mark optional features in the software product line with `#ifdef` annotations. Considering the actual code transformation, we incrementally integrated the most similar games (in terms of identified code clones) into a common platform. To this end, we relied on the pairwise diffing of Melt that helped us to compare classes and to identify the exact differences between games. Initially, we focused on integrating two games (i.e., ApoDice and ApoSnake) and getting them to build and run, allowing us to test our setup and the source code. So, we first integrated only the common base and the main differences of these two games in few features. Afterwards, we annotated additional features to increase the variability of the software product line. We remark that we could still not get ApoSnake to run. After we ensured that we could build and run ApoDice as a variant of the software product line, we continued with integrating the other legacy games, in the following order: ApoClock, ApoMono, and MyTreasure. Unfortunately, we repeatedly had problems with building and running the configured games in Android Studio, which we could not fully resolve.

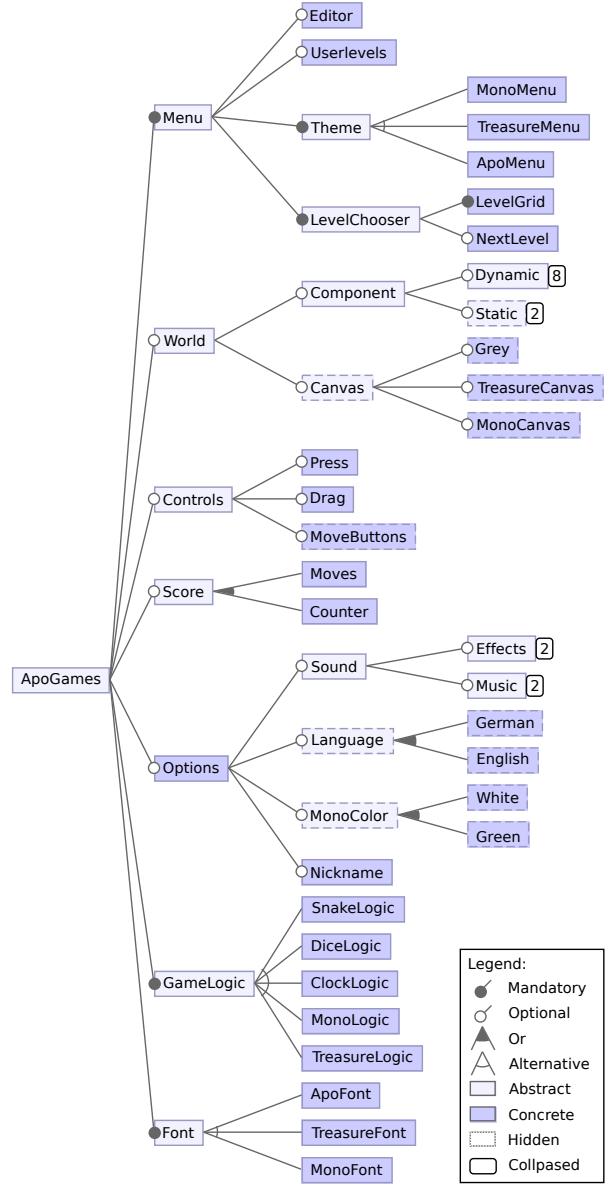
During our the transformation, we constantly performed quality assurance to maintain the quality of the resulting software product line. For this purpose, we configured the games in FeatureIDE to then build and execute them in Android Studio. Besides testing for flaws in the code that may results in a bug or errors, we also compared the variants to the legacy games. With this comparison, we aimed to identify any unintended differences, for example, because we used Antenna incorrectly or lost functionalities.

## 3 RESULTS

In this section, we describe the artifacts we created. We summarize the characteristics of the feature model and source code of the software product line compared to the legacy games in Table 2.

<sup>3</sup>[https://pmd.github.io/pmd/pmd\\_userdocs\\_cpd](https://pmd.github.io/pmd/pmd_userdocs_cpd)

<sup>4</sup><http://meldmerge.org/>



**Figure 1: Excerpt of the feature model of the extracted software product line. Hidden features are not yet configurable.**

**Feature Model.** In total, we identified 61 features, of which 42 are concrete features, and we implemented 27 of them in the software product line. Our implementation is a reduce set of all features, due to time constraints and unexpected efforts (cf. Section 4). We defined a group of child feature below each feature that required game-specific adaptations. As we can see in Figure 1, we defined few mandatory features (i.e., *GameLogic*, *Menu*, *Theme*, *LevelGrid*, *Font*), which are needed to set up the rough architecture of a game. In particular, the feature *GameLogic* is the most basic one that contains the more advanced game algorithms. Other features are optional and provide additional functionalities that were available in some games, such as an *Editor* and *Options*.

We defined 14 cross-tree constraints in the model. These ensure that a user can only configure a game that will work, even though not all configurations will result in a reasonable game. Thus, these constraints ensure that the right game elements (like a branding), have to be selected together. For example, *TreasureGameLogic* is implied if the corresponding *TreasureMenu* is selected. We included hidden features that we did not annotate with Antenna, yet.

**Source Code.** Considering the implementation, we migrated all selected Apo-Games into a common platform, but did not annotate all optional features. However, we achieved a considerable reduction in code size of approximately 51%. We remark that this number results from large restructurings and merges during the transformation that reduced the redundant code clones, but is not completely correct, due to tooling differences and the fact that Antenna uses comments for its annotations. Moreover, we could reduce the number of files from a total of 117 to 43. Considering that the largest legacy game had 28 classes on its own, these statistics show that there is an increase in the architectural complexity for the software product line, which we expected as it integrates multiple games. Nonetheless, the reduced size highlights the potential for reuse even for smaller cloned systems, such as Android games.

## 4 LESSONS LEARNED

During the analysis and transformation of the Android games, we experienced some challenges that we discuss in the following. Some points are strongly connected to Android and technical issues during software evolution, both of which may appear in any organization or open-source project at some point.

**Dissimilarity due to Libraries.** When we analyzed the architectures of the games, we expected that the games would be quite similar, due to the similar structure. However, despite ApoDice and ApoClock sharing almost all classes and the same look-and-feel, the actual implementations differ quite a lot. We found that these differences are due to the game engine that was used in two different versions, resulting in changes in, for example, the rendering technique. In our experience, such external libraries cannot only conflict expectations, but were also hampering variability.

**Unexpected Effort of Branding.** All of the games share some common elements, such as menus. We expected that these would comprise a lot of cloned code and only small variations to change the look-and-feel of individual games. However, we experienced that these views were fundamentally different in their source code. Thus, we re-designed common features and implemented them from scratch, only introducing branding as child features. This way, we reduced the redundancy that may have been in the software product line, but it required unexpectedly high effort and we had to stop at some point to focus on integrating features.

**Deciding about Features.** We experienced that it was challenging to decide whether we should try to extract a common feature or not. As aforementioned, merging can be an effortful task, because some features differ heavily in the legacy games, for instance, due to the different game types and evolution (e.g., the external game engine). After the migration, we are still not sure if extracting a software product line from the Android games was worth the effort. In our opinion, the systems need to be much closer to each other (i.e., not outliers [15]) and this may be a common problem with games.

**Readability of Source Code.** A common argument against annotations for variability is that they obfuscate the source code, which can become lengthy and hard to read [1, 20]. We experienced exactly this issue while transforming the variants, as we had to merge various implementations of the same method into one. This problem can be tackled with additional refactoring, but it still hampers the integration of variants and is a source for errors.

**Available Tooling.** We used FeatureIDE for most parts of our case study and found it most useful, too. In the initial phase, we identified other tools that could support the analysis of the legacy games. However, few of such research tools were available in a usable state or provided more help compared to other open-source solutions that we used. We found several specialized tools that could have been helpful with additional artifacts besides the source code. Overall, we found no established tool for software product lines that sufficiently supports the extraction process that we applied.

## 5 THREATS TO VALIDITY

**Internal Validity.** Unfortunately, we could not cooperate with the original developer of the Apo-Games for this case study. Thus, we had to analyze the legacy games and scope features ourselves. While we were careful and checked the results constantly, other researchers or practitioners would arguably still derive slightly different results. However, we could run and execute the legacy games form our software product line and our experiences as well as artifacts are nonetheless valuable baselines for future work.

**External Validity.** We had only access to five rather small Android games to migrate them. So, our process and the results may not be fully comparable to real-world scenarios. Still, considering that only few subject systems for the extraction of software product lines are available, the Apo-Games are a valuable dataset. Moreover, our case study already revealed important insights into problems that will only increase for larger systems, wherefore our experiences and artifacts are helpful for future research and practitioners.

## 6 CONCLUSION

In this paper, we described a case study in which we extracted a software product line from five Android games of the Apo-Games dataset. From this case study, we contributed the resulting artifacts (i.e., feature model, implemented software product line) and reported our experiences. Our main insight is that merging clones is a challenging process, even if we rely on annotation-based approaches that are often argued to only require simple additions of variability into the code [1]. However, locating, scoping, and merging of features remained challenging tasks that require considerable efforts. These problems are more conceptual and independent of the used variability mechanism.

In future work, we want to extend and refine our artifacts to enable researchers to use them as suitable ground truths for evaluations. Moreover, we plan to conduct further case studies to verify our insights. Especially, we aim to understand what factors (e.g., the variability mechanism) have what impact on processes and efforts.

**Acknowledgments.** This work is supported by the ITEA project REVaMP<sup>2</sup> funded by Vinnova Sweden (2016-02804), and the Swedish Research Council Vetenskapsrådet (257822902). We thank Jennifer Horkoff for valuable comments on this work.

## REFERENCES

- [1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.
- [2] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Reengineering Legacy Applications into Software Product Lines: A Systematic Mapping. *Empirical Software Engineering* 22, 6 (2017), 2972–3016.
- [3] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 7:1–7:8.
- [4] John Businge, Openja Moses, Sarah Nadi, Engineer Bainomugisha, and Thorsten Berger. 2018. Clone-Based Variability Management in the Android Ecosystem. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 625–634.
- [5] Paul C. Clements and Linda M. Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley.
- [6] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34.
- [7] Slawomir Duszynski, Jens Knodel, and Martin Becker. 2011. Analyzing the Source Code of Multiple Software Variants for Reuse Potential. In *Working Conference on Reverse Engineering (WCRE)*. IEEE, 303–307.
- [8] Peter Knauber, Jesus Bermejo, Günter Böckle, Julio C. S. do Prado Leite, Frank J. van der Linden, Linda M. Northrop, Michael Stark, and David M. Weiss. 2002. Quantifying Product Line Benefits. In *International Workshop on Software Product-Family Engineering (PFE)*. Springer, 155–163.
- [9] Charles W. Krueger. 2002. Easing the Transition to Software Mass Customization. In *International Workshop on Software Product-Family Engineering (PFE)*. Springer, 282–293.
- [10] Jacob Krüger, Thorsten Berger, and Thomas Leich. 2019. *Software Engineering for Variability Intensive Systems*. CRC Press, Chapter Features and How to Find Them: A Survey of Manual Feature Location, 153–172.
- [11] Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. 2016. Extracting Software Product Lines: A Cost Estimation Perspective. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 354–361.
- [12] Jacob Krüger, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. 2018. Apo-Games - A Case Study for Reverse Engineering Variability from Cloned Java Variants. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 251–256.
- [13] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is my Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software* 152 (2019), 239–253.
- [14] Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2017. Finding Lost Features in Cloned Systems. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 65–72.
- [15] Crescencio Lima, Wesley K. G. Assunção, Jabier Martinez, William Mendonça, Ivan C Machado, and Christina F. G. Chavez. 2019. Product Line Architecture Recovery with Outlier Filtering in Software Families: The Apo-Games Case Study. *Journal of the Brazilian Computer Society* 25, 1 (2019), 7.
- [16] Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. 2017. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 38–41.
- [17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.
- [18] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. 2016. Synchronizing Software Variants with Variantsync. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 329–332.
- [19] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering*. Springer, 29–58.
- [20] Janet Siegmund, Christian Kästner, Jörg Liebig, and Sven Apel. 2012. Comparing Program Comprehension of Physically and Virtually Separated Concerns. In *International Workshop on Feature-Oriented Software Development*. ACM, 17–24.
- [21] Ştefan Stănicălescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 151–160.
- [22] Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *International Systems and Software Product Line Conference (SPLC)*. ACM. Accepted.
- [23] Frank van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action*. Springer.
- [24] Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig. 2006. Assessing Merge Potential of Existing Engine Control Systems into a Product Line. In *International Workshop on Software Engineering for Automotive Systems (SEAS)*. ACM, 61–67.

# DNA as Features: Organic Software Product Lines

Mikaela Cashman  
Computer Science  
Iowa State University  
Ames, IA, USA  
mcashman@iastate.edu

Justin Firestone  
Computer Science & Engineering  
University of Nebraska-Lincoln  
Lincoln, NE, USA  
jfiresto@cse.unl.edu

Myra B. Cohen  
Computer Science  
Iowa State University  
Ames, IA, USA  
mcohen@iastate.edu

Thammasak Thianniwit  
School of Information Technology  
Suranaree University of Technology  
Nakhon Ratchasima, Thailand  
thammasak@sut.ac.th

Wei Niu  
Chem. & Biomolecular Engineering  
University of Nebraska-Lincoln  
Lincoln, NE, USA  
wniu2@unl.edu

## ABSTRACT

Software product line engineering is a best practice for managing reuse in families of software systems. In this work, we explore the use of product line engineering in the emerging programming domain of synthetic biology. In synthetic biology, living organisms are programmed to perform new functions or improve existing functions. These programs are designed and constructed using small building blocks made out of DNA. We conjecture that there are families of products that consist of common and variable DNA parts, and we can leverage product line engineering to help synthetic biologists build, evolve, and reuse these programs. As a first step towards this goal, we perform a domain engineering case study that leverages an open-source repository of more than 45,000 reusable DNA parts. We are able to identify features and their related artifacts, all of which can be composed to make different programs. We demonstrate that we can successfully build feature models representing families for two commonly engineered functions. We then analyze an existing synthetic biology case study and demonstrate how product line engineering can be beneficial in this domain.

## CCS CONCEPTS

- Software and its engineering → Software product lines;
- Applied computing → Systems biology.

## KEYWORDS

Software Product Lines, BioBricks, Synthetic Biology

### ACM Reference Format:

Mikaela Cashman, Justin Firestone, Myra B. Cohen, Thammasak Thianniwit, and Wei Niu. 2019. DNA as Features: Organic Software Product Lines. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3336294.3336298>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336298>

## 1 INTRODUCTION

Today's software development practices are dominated by reusability, with practitioners sharing plug-and-play modules that can be applied in a multitude of different applications. As part of this movement, open-source repositories such as GitHub have emerged as marketplaces where developers find libraries and other reusable components. At the same time, software product line (SPL) engineering has become a best practice for modeling and managing families of software systems. The SPL community has turned to open-source systems and used the concepts of commonality and variability to define open-source product lines [31]. Recently, Montalvillo and Díaz have proposed techniques to aid SPL practices within GitHub [25]. It seems natural to look at other emerging open-source marketplaces to understand whether SPL development can provide benefits to those communities as well.

Synthetic biology, the practice of engineering living organisms by modifying their DNA, has advanced rapidly over the last 30 years [8]. It is being used for sensing heavy metals for pollution mitigation [5], development of synthetic biofuels [40], engineering cells to communicate and produce bodily tissues [29], emerging medical applications [22, 38], and basic computational purposes [12]. Synthetic biologists design new functionality, encode this in DNA strands, and insert the new DNA *part* into a living organism such as the common K-12 strain of the bacteria *Escherichia coli* (*E. coli*). As the organism reproduces, it replicates the new DNA along with its native code and builds proteins that perform the encoded functionality. In essence, the biochemist is *programming* the organism to behave in a new way. Hence we call these *organic programs*.

As DNA strands have become easy to engineer by simply purchasing a desired sequence, the field of synthetic biology has rapidly grown. For instance, each year 300+ teams of students (high school through graduate) compete in the International Genetically Engineered Machine (iGEM) Competition. Teams build genetically engineered systems to solve real-world problems [20]. Students are required to submit the engineered parts along with their designs and experimental results back into an open-source collection of DNA parts called *BioBricks*. This BioBrick repository (called The Registry of Standard Parts) [21] contains over 45,000 DNA parts and can be viewed as a Git repository for DNA. In this paper we utilize this as our exemplar system, but we note that companies

and other institutions are likely building their own commercial instances of this type of repository.

DNA parts are often advertised as “LEGO” pieces that can be combined in many ways to form new genetic devices. However, these LEGO pieces come with no “Building Instructions.” An engineer begins a project by developing a blueprint for the organic program they want to build. They will have a general plan for the type of features they want their system to have (such as a part that produces a fluorescent protein or expresses a gene in the presence of a particular chemical). To bring this creation to fruition they must find the corresponding parts in a repository or in the literature. They then build the associated working organic system following an architecture that merges the features together. However, this architecture can require significant domain knowledge to develop. Rather than expecting engineers to create architectures from scratch, we hypothesize that SPL engineering can be used.

In this work we explore the use of product line engineering with the goal of helping developers in synthetic biology. We conjecture that there are families of products that consist of common and variable DNA parts, just as we see in other open-source repositories. We call these *organic software product lines*. We conduct a case study to evaluate this claim and lay the foundations for merging software product line engineering and synthetic biology.

The contributions of this work are:

- (1) A mapping of SPL engineering to the domain of synthetic biology resulting in organic software product lines;
- (2) A case study demonstrating the potential reuse and existence of both commonality and variability in the BioBricks repository and showing that we can build feature models that have potential to help synthetic biologists.

In the next section we present background on synthetic biology and a motivating example. We then propose the notion of an organic SPL (Section 3). We follow this with our case study (Section 4), results (Section 5), and discussion (Section 6). We present related work in Section 7, and end with conclusions and future work.

## 2 BACKGROUND

Synthetic biology has been defined as a process “to design new, or modify existing, organisms to produce biological systems with new or enhanced functionality according to quantifiable design criteria” [1]. Part of this definition emphasizes design and quantification. It follows that we can view synthetic biology as a programming discipline. It begins with a model, which is then implemented into strands of DNA that are inserted into a living cell. We can view the organism as the compiler that takes the DNA and translates it to machine level code, creating proteins that the organism uses to perform different functions. Just as machine code is written in 1s and 0s, biology is written in the four DNA bases adenine (A), thymine (T), cytosine (C), and guanine (G).

This analogy of *programming biology* is not a new concept. There is even a programming language called the Synthetic Biology Open Language (SBOL) which defines a common way to represent biological designs [30]. Researchers have used synthetic biology to create a context-free grammar using BioBricks [7], automated design of genetic circuits with NOT/NOR gates [26], and bacterial networks to use DNA for data storage [6, 32]. There are also several examples

of organic programs inspired by classic computer science constructs such as a genetic oscillator [13], a genetic toggle switch [17], and a time-delay circuit [39].

At its core, synthetic biology breaks down a biological process into smaller functions, each of which can be represented by a DNA *part*. These parts can be put together in various combinations to make new functions. The largest open-source repository of DNA parts is called the Registry of Standard Parts [21] (Registry for short). Parts have been contributed to this registry through the iGEM Competition. Although participants from iGEM are the most frequently documented users of the repository, anyone can use it to find appropriate parts for their designs.

iGEM describes itself as a competition where teams “design, build, test, and measure a system of their own design using interchangeable biological parts and standard molecular biology techniques.” Each year about 6,000 students participate, designing projects which often address various regional problems such as pollution mitigation. Teams are judged by community experts and can be awarded a medal (bronze, silver, and gold) corresponding to the impact and contributions of their project. A gold medal team must achieve several goals such as modeling their project, demonstrating their work through experimentation, collaborating with other teams, addressing safety concerns, improving pre-existing parts or projects, and contributing new parts.

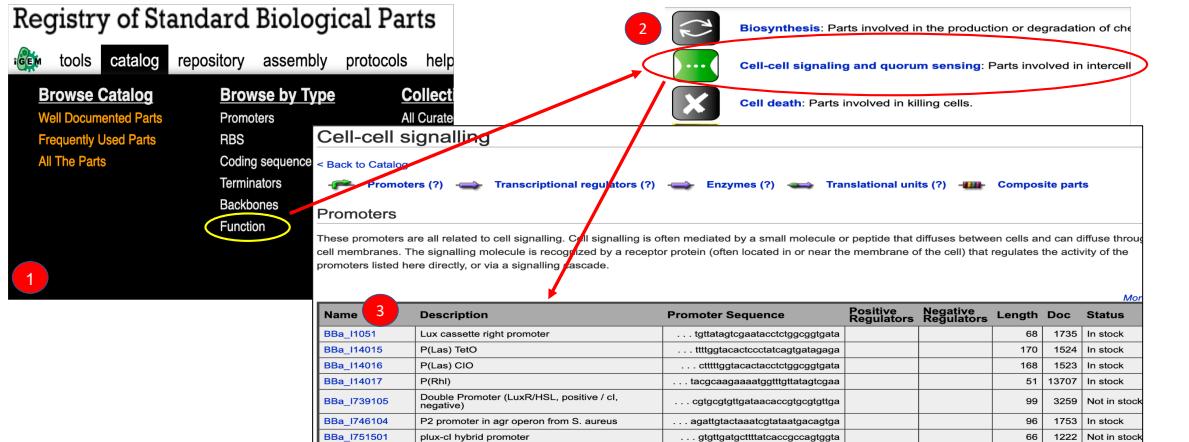
### 2.1 Motivating Example

We next present a motivating example derived from our case study demonstrating the potential of SPL engineering in this domain.

Cell-to-cell signaling is a common function of synthetic biology. It represents a key communication mechanism for cellular organisms. A *sender* organism communicates with a *receiver* organism which responds to the signal by emitting some chemical *reporter*. Suppose an engineer wants to build a cell-to-cell signaling system from scratch. If the engineer has no additional resources other than an online repository and his or her knowledge of synthetic biology, then this analogy is similar to someone searching GitHub for code that performs a particular function. The user can search the Registry for “signalling” and they will be redirected to a single page with a list of parts. It consists of eight senders, 11 receivers, and 464 “other” parts. If, however, the user searches with U.S. English convention for “signaling” the query returns 789 hits, each with its own page to investigate. It is important to note that not all of these page hits link to parts actually involved in cell-to-cell signaling. Some are pages that simply have the word “signaling” in them. This demonstrates the difficulty of any free-text search.

An alternate strategy is to search based on function using the field “Browse parts and devices by function.” Figure 1, steps #1 and #2, show this in the Registry. There are 10 functions listed including “Cell-to-cell signaling and quorum sensing.” Parts are sorted (Figure 1 - step #3) into various lower-level categories on this page. Most are basic parts that include: 39 promoters, 13 transcriptional regulators, 12 enzymes, and 21 translational units. There is also a separate list of 138 composite (or aggregate) parts.

Another strategy would be to search for previous projects that built a cell-to-cell signaling system. For example, one might locate



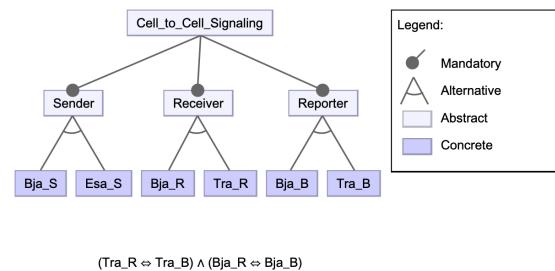
**Figure 1: Browse by function → Cell-to-cell signaling**

the 2017 iGEM team from Arizona State University (ASU) [2]. Looking on this team's web page would lead the user to find models for 30 composite parts for cell-to-cell signaling. While this is an improvement over the prior approaches and provides a roadmap to build the system (along with results of the study), it is limited to the 30 products that the ASU team chose to use in their experiments. As we will show, there are many more ways to build a cell-to-cell signaling system.

What if, instead of starting from scratch, the synthetic biologist begins this process with the feature model shown in Figure 2 (a subset of a feature model in our study)? From this model the user immediately can see the architecture of their system. First, they learn that any cell-to-cell signaling system requires three basic parts: a sender, a receiver, and a reporter. Instead of having to look at hundreds of possible parts, the user can also see there are only two possible parts for each of the three components. The user also notices a constraint in this model, so they will also not waste time testing combinations of features which are incompatible.

If users wanted to test the effectiveness of various receivers in this system, they could slice this model to get a specific set of products. They could also design experiments to test products that have not yet been analyzed in the laboratory. Once they complete their experiments, users could add their results back to the Registry as annotations. This is a small example, but it demonstrates how product line engineering could help users construct valid cell-to-cell signaling programs.

As further motivation, if we return to the ASU team's experiments, one of their goals was to investigate the crosstalk, or the interactions between various parts. To test this, they designed experiments with multiple combinations of senders and receivers. Without realizing it, they defined a family of products for cell-to-cell signaling and evaluated the individual products. If they were working with a feature model they would have been able to: 1) efficiently sample the product space; 2) know how much of the space they explored; and 3) add constraints when they found crosstalk between parts. They could then annotate the feature models and create assets to describe their findings which could be used by another team working on a similar project. In essence, they could



**Figure 2: Example feature model for cell-to-cell signaling.**

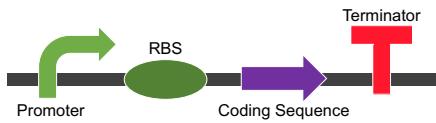
leverage the power of SPLs. It is this motivation that leads us to propose organic software product lines.

### 3 ORGANIC SOFTWARE PRODUCT LINES

Not long ago, the SPL community asked if open-source applications such as the Linux kernel should be considered product lines given that they are not managed and developed in the traditional manner [23, 31]. This has led to a broader view of SPLs. We ask the same question now of organic programs. Is there a mapping between traditional software product line engineering and synthetic biology that allows for managed development and reuse?

As Clements and Northrop state, the output of domain engineering should contain (1) a product line scope, (2) a set of core assets, and (3) a production plan [10]. This feeds into application engineering, which uses the production plan and scope to build and test individual products. Our focus in this work is primarily domain engineering, however we touch upon application engineering in the context of organic programs in the case study.

**Assets.** To begin we need to identify what constitutes a core asset for this domain. Assets in traditional product lines can include a software architecture, reusable software components, performance models, test plans and test cases, as well as other design documents. In organic programs, we see similar elements, discussed next. SBOL (or a similar representation) is used to define the functionality of a snippet of DNA code. This serves as an important design



**Figure 3: SBOL model of a transcription unit.** This composite part is composed of four basic DNA parts: The promoter (also called a regulator), ribosome binding site (RBS), coding sequence, and terminator.

document for individual features. SBOL models can be composed and aggregated leading to composite models. The SBOL model for the simplest, stand-alone functional biological unit (called a *transcriptional unit*) can be seen in Figure 3. It is composed of four basic DNA parts: the promoter, ribosome binding site, coding sequence, and terminator.

The DNA sequence is the reusable software component. Like code it is not tangible, but must be implemented as a program and compiled to a machine level (or byte-code level) representation. DNA can be synthesized into a physical strand which can be inserted into a compiler (the living organism) for translation to machine level code (via the biological processes of transcription and translation of DNA via RNA into proteins). Other assets such as test cases and test plans can be constructed which define either laboratory experiments or virtual simulations. Both lead to evaluation of the program’s expected, versus observed, functionality. Additional assets in the form of design documents and documentation can be provided such as safety cases [16] and higher level system architecture (e.g., GenoCAD [7]).

**Domain Engineering.** During domain engineering the engineer defines the product line scope by choosing a family of behavior such as a type of molecular communication. He or she also defines the common and variable features and their relationships. An example of commonality is the transcription unit (Figure 3). The specific choices for promoters and binding sites defines the variability. When inserted into their host organisms at specific binding sites, the sets of DNA sequences define unique sets of *products*. Last, a production plan can be created in combination with a feature model and constraints. The feature model and constraints show how the DNA parts, as *features*, form a family of products, along with experimental notes on expected environmental conditions or other assumptions that are required for the program to run correctly.

**Application Engineering.** Application engineering involves combining the expected parts using standard DNA cloning techniques for insertion into a living organism [28]. Just as with traditional product lines, it is up to the engineer to adhere to constraints and only compose products defined in the feature model, otherwise unexpected behavior may occur. As in traditional software, some constraints may be hard-coded into the program, while some may represent a domain expectation instead.

Thüm *et al.* [36] in their survey of product lines discuss different implementations of product lines and analyses that can be performed at both the family and variability level. We believe organic product lines are just another type of implementation and we can utilize common analyses and techniques from SPL engineering in this domain. We study that next.

## 4 CASE STUDY

We conducted a case study to evaluate the feasibility of using product line engineering in synthetic biology. Supplemental data for this study can be found on our website.<sup>1</sup> We ask the following research questions:

**RQ1:** Does a DNA repository have the characteristics of a software product line?

**RQ2:** Can we build feature models representing families of products from an existing DNA Repository?

**RQ3:** Can SPL engineering provide useful analyses for developers of organic programs?

### 4.1 Subject Repository

We use as our subject repository the *Registry of Standard Biological Parts* (herein referred to as the BioBrick repository) as described in Section 2. We choose this subject as it is the largest open-source DNA repository and continues to grow (on average 2,995 parts are added each year).

For this work we define a *feature* simply as a BioBrick part. A *product* is the compilation of multiple basic BioBricks that together perform a cohesive function.

### 4.2 Data Collection

To study the core assets of the system, we use the BioBrick API to pull data for all parts up through December of 2018, consisting of 47,934 entries [19]. Each entry contains information such as the part\_id, part\_name, part\_type, uses, status, and creation\_date. *Usage* in this case is defined by counting how many times a part has been requested by a community user. This can tell us how useful a part is to the community.

### 4.3 Study Objects

For RQ1 we use the BioBrick repository. For RQ2 we select two biological functions in which to build a feature model. The BioBrick repository has sorted parts into ten common biological functions: biosafety, biosynthesis, cell-to-cell signaling and quorum sensing, cell death, coliroid, conjugation, motility and chemotaxis, odor production and sensing, DNA recombinations, and viral vectors. We select the two functions with the greatest number of parts (biosafety and cell-to-cell signaling). Biosafety has several subcategories, we choose the subcategory with the most parts –kill switch. For RQ3 we selected a 2017 iGEM team from Arizona State University [2] whose project is a subcategory of cell-to-cell signaling.

**4.3.1 Kill Switch.** A kill switch is a safety mechanism which triggers cellular death, typically by engineering cells to produce proteins which destroy cellular membranes. Common triggers include exposure to specific chemicals, temperature ranges, pH levels, or frequencies of light. To build the kill switch feature model we manually reviewed all of the wiki pages from the 110 teams who earned a gold medal in the 2017 iGEM competition [16]. Only 14 mentioned some type of kill switch in their design. We use those pages. The exact set of teams is listed on our website.

<sup>1</sup><https://sites.google.com/view/splc-dnafeatures>

**Table 1: Part types for all BioBrick parts in the repository.**

part_type	# parts	part_type	# parts
Coding	10,265	RBS	769
Composite	9,966	Primer	685
Regulatory	4,165	Plasmid	681
Intermediate	3,506	Project	656
Generator	2,425	Terminator	518
Reporter	2,310	Signalling	511
Device	2,277	Plasmid_Backbone	454
DNA	1,717	Tag	385
Other	1,419	Scar	121
Measurement	1,162	Inverter	117
RNA	976	Cell	75
Protein_Domain	917	T7	57
Translational_Unit	880	Conjugation	51
Temporary	866	Promoter	3

**4.3.2 Cell-to-Cell Signaling.** Cell-to-cell signaling is a key cellular communication mechanism by means of secreting and sensing small molecules or peptides/proteins between a sender and a receiver. To build a feature model for cell-to-cell signaling, we forward-engineered the parts listed under the cell-to-cell signaling category of the BioBrick repository. We employed basic knowledge of the structure of a cell-to-cell signaling category and sorted parts by their features.

There are 39 promoters, 13 transcriptional regulators, 10 biosynthesis enzymes, 2 degradation enzymes, 21 transitional units, and 138 composite parts in this category. Since we want to map systems down to the lowest level of feature we ignore the composite parts.

## 5 RESULTS

In this section we answer each of our research questions in turn. In RQ1 we quantify assets from the BioBrick repository and explore both commonality and variability between products. We focus on domain engineering in RQ2 and RQ3. We move to application engineering in RQ3.

### 5.1 RQ1 Does the BioBrick repository have the characteristics of a software product line?

In Section 3 we defined organic software product lines in terms of SPL engineering concepts. We start with the core assets, the code. There are 47,934 BioBrick parts at the time of this publication. Table 1 shows the counts of parts by function. The largest category, *coding*, has over 10,000 parts. These are sequences that encode specific proteins. The second most frequent category (9,966) is *composite part*. A composite part is composed of two or more basic parts (*i.e.*, an aggregate class or function). All of the top ten categories have more than 1,000 parts. We can consider these reusable assets for building products.

We next analyzed the *use count* for each part. The use count specifies how many times a request for the part was made by an external user. This is similar to a GitHub checkout. Table 2 displays this data. We can see that the majority of parts (about 71%) are never requested. Approximately 27% are used between one and

**Table 2: BioBrick use counts - # user requests.**

# of Uses	# of Parts
0	34,091 (71.12%)
1-10	13,117 (27.36%)
11-50	602 (1.26%)
51-100	61 (0.13%)
101+	63 (0.13%)

**Table 3: Assets present in 100 random composite parts.**

Asset	SBOL model	DNA sequence	Textual description	Experimental results
% parts	82%	95%	90%	24%

ten times. Then we see a small percentage of parts (under 2%) that are used more than 11 times. This demonstrates the repository consists of many reusable core assets. The parts with high use may show potential commonality between projects, and the parts with lower use may represent potential variability. We leave a complete analysis as future work. We see a similar phenomenon in traditional software repositories with a large abundance of code, but a comparatively small number of highly used modules [42].

Each part’s web page in the BioBrick repository can contain several additional assets including the SBOL model, the raw DNA sequence, a part description in plaintext, and results from experimentation from iGEM teams. All of these assets may be useful to a user interested in how a part can fit into their construct. We randomly sampled 100 composite parts from the repository and identified whether they had these assets. Table 3 shows the results. 82% of parts included the SBOL format. Most of them included the raw DNA sequence (95%) and a basic textual description (90%). Only 24% of parts included any additional experimental results.

We now focus on two aspects of SPLs that we would expect to see in practice:

**Variability.** To examine variability we focus on the transcription unit, the most basic function (see Figure 3). There are 4,165 promoters, 769 RBSs, 10,265 coding sequences, and 518 terminators. If we underestimate the possible product space by counting one of each part (a standard practice is to use two terminators which will increase the space by a large factor) we have on the order of  $1.7 \times 10^{13}$  (17 trillion) products representing transcription units.

There are also 9,966 parts labeled as composite in the repository (meaning they were built from basic parts and added back into the repository). Each represents one customized product built from the core components, again showing variability.

**Commonality.** Not every product is completely distinct from others. Products will share certain common features with other products in their biological functionality. There are ten functional categories listed in the BioBrick repository. Each of these categories can represent one set of products, and they will share common architectural elements. Two examples of this include: (1) a kill switch will always have a trigger and an effect; and (2) a cell-to-cell signaling system will always have a sender, receiver, and reporter. In RQ3 we examine a real family of products that has 15 common features.

## 5.2 RQ2 Can we build a feature model from the BioBrick Repository?

For this research question we built two different feature models for two different functions in the BioBricks repository.

**5.2.1 Kill Switch.** Based on manual review of 2017 iGEM teams who earned a gold medal, we built the feature model shown in Figure 4. The *trigger type* (left side) is the most interesting because synthetic biologists will want to engineer kill switches to activate only under certain conditions. The kill switches we reviewed can be triggered under several different conditions: temperature ranges; presence or absence of specific chemicals; low pH levels; or exposure to specific frequencies of natural light. Each of those trigger conditions ends in leaf nodes which are the BioBrick IDs correlated to specific DNA sequences. The promoters, RBSs, coding sequences, and terminators show which BioBricks can be used to complete a transcription unit for a kill switch. The *visualization* branch is optional. It provides visual evidence that the switch is working through production of fluorescent proteins. This model represents 882 valid (different) kill switches.

**5.2.2 Cell-to-Cell Signaling.** Figure 5 shows the overall cell-to-cell signaling feature model we constructed. Recall that a basic cell-to-cell signaling system has three different of transcriptional units (sender, receiver, reporter). The feature model follows a hierarchical model with variation points at the transcriptional unit level as a sub-feature model. Because the full cell-to-cell signaling model is too large to show here, we visually present only some of these sub-feature models and describe the rest in text (the complete model is on our supplementary website).

**Promoter:** One promoter is needed for the sender, one for the receiver, and one for the reporter. A promoter has three features: *constitutiveness*, *activation*, and *repression*. A promoter may have a different level of constitutive expression (meaning it expresses on its own, without being activated by any protein). We found parts that categorize this as *weak*, *medium*, or *strong*. A promoter can also be activated (increased expression) by a protein. We identified four proteins listed under cell-to-cell signaling promoters. We identified seven proteins that could be used for repression.

We represent the three features of a promoter (constitutive, activation, repression) with an OR relationship. Each of the parts below these features have an Alternative relationship. In our model one promoter alone has 159 possible configurations. The model for the receiver promoter is expanded and can be seen in Figure 5.

**RBS:** The next high-level feature is the RBS. This part will have the same variation in the sender, receiver, and reporter. Since the cell-to-cell signaling catalog does not include RBS parts, we looked at all RBS parts. They are sorted into different collections, so we use the community collection. The functional differences between them is in their protein expression level (“strength”). This feature in the SPL has eight possible configurations.

**Coding Sequence (Protein):** The next part is the coding sequence. We limit this model to only coding sequences which encode for creation of specific proteins. We have identified five proteins in the cell-to-cell signaling catalog. In addition to a protein, a *function* is required to be chosen, either *transcriptional regulation* or an *enzyme*. An enzyme can either be for *biosynthesis* or *degradation*. There is

also an optional *LVA tag* which reduces the proteins’ half-life. The coding sequence parts have 30 possible configurations. This model is shown in Figure 6.

**Coding Sequence (Reporter):** The other type of coding sequence we model represents the encoding of the reporter’s signal. We identify four possible behavioral responses: green fluorescence, biofilm production, antibiotic production, and a kill switch.

**Terminator:** The final part is the terminator. There are no terminators listed in the cell-to-cell signaling catalog, so we look at all terminators in the repository. Terminators can be in the forward direction, reverse direction, or be bidirectional. We only consider forward directional terminators in this model. In the BioBrick catalog there are 24 terminators available. It is common to choose two terminators to ensure transcription stops, so in our model we allow choosing one or two (a {1,2} OR relation) terminators. The terminator parts allow 300 possible configurations. Since our model was too large for FeatureIDE to calculate the set of products we use FaMa [4]. The sender and receiver each have 11,448,000 products. The reporter has 5,724,000 products. The total number of products is  $7.5 \times 10^{20}$ .

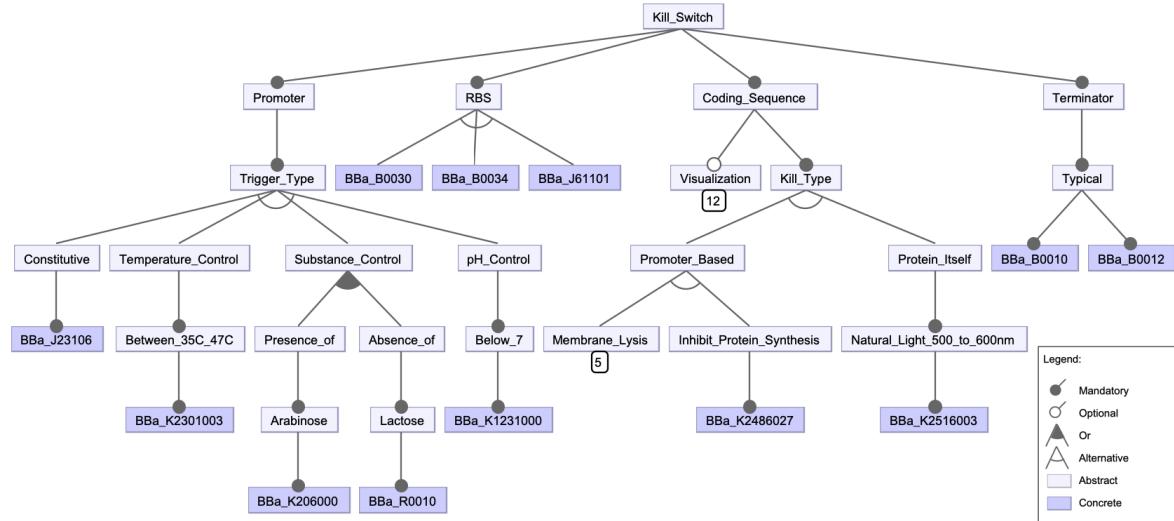
## 5.3 RQ3: Can SPL engineering provide useful analyses for organic programs?

For this research question we ask questions related to application engineering. We use the 2017 iGEM team project from Arizona State University [2, 33] introduced in our motivating example. Because this team built a quorum sensing network, which is a type of cell-to-cell signaling, it helps us validate the results from RQ1 from an application engineering perspective.

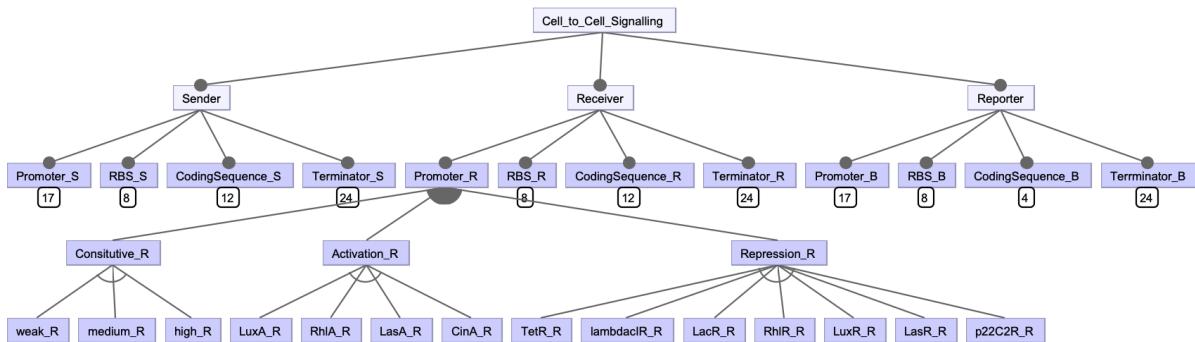
In quorum sensing, there are two groups of organisms: the first group acts as the sender, and the second as the receiver. The receiver will exhibit a type of behavioral response at a certain concentration (a quorum) signaled by the presence of both these proteins. For example, the receiving organism may turn green when enough of the sending organism’s signal is sensed in the system.

The choice of protein for the sender and the receiver plays an important role. Some combinations of proteins cause crosstalk for the receiver which can render the system inefficient or even useless. As stated on the ASU team’s project web page: “Knowing the rates of induction also allows for greater precision when designing genetic circuits.” The team chose ten different proteins for the sender (Aub, Bja, Bra, Cer, Esa, Las, Lux, Rhl, Rpa, Sin), three different proteins for the receiver (Las, Lux, Tra), and three different proteins for the reporter (Las, Lux, Tra).

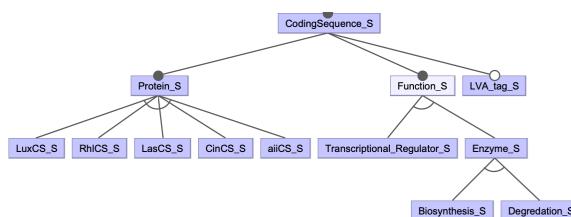
**5.3.1 Providing a Broader View of the Product Space.** Though perhaps unintentionally, the ASU team’s project represents a feature model. We formalize it by manually reverse-engineering their project feature model from their web page. Figures 7 and 8 show the resulting sender and receiver models respectively (herein referred to together as the *ASU model*). This model contains the high-level features: sender and receiver. The receiver contains both a regulator and reporter. Each feature has four basic parts (promoter, RBS, coding sequence, terminator) and the sender has an extra feature to incorporate a red fluorescence. Many of the features are mandatory. The points of variability lie in the sender’s coding sequence, the regulator’s coding sequence, and the reporter’s promoter. This model



**Figure 4: A feature model for a kill switch. An integer on a leaf represents how many nodes are in their subtrees (obfuscated).**



**Figure 5: Top levels of the cell-to-cell signaling feature model.**



**Figure 6: Sub-feature model of a protein coding sequence.**

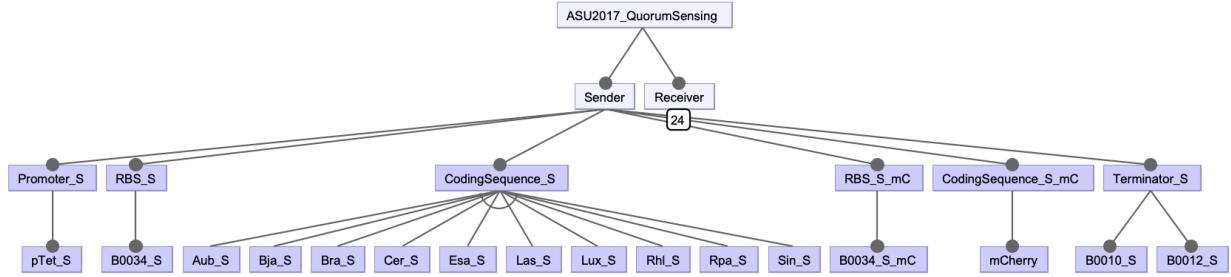
represents a total of 90 products. To conduct their experiments, the ASU team added a constraint between the regulator's coding sequence and the reporter's promoter (they are required to be the same). Thus their experiment tested 30 of these products in the laboratory. We call this the *ASU experiment model*.

If we compare this ASU model to the reverse engineered model presented in Section 5.2.2 (we call this the *cell-to-cell signaling model*), the ASU model is not a direct subset of the cell-to-cell

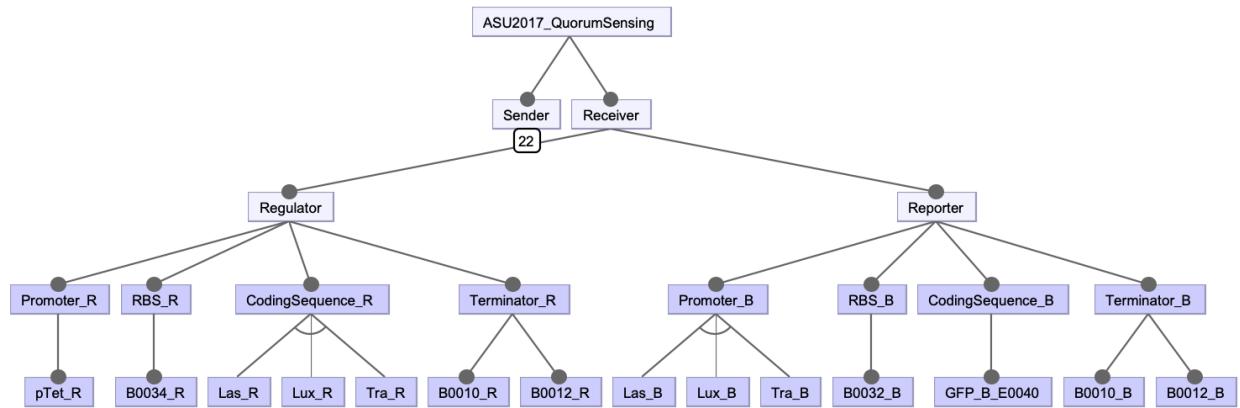
signaling model. In practice it should be. We would have expected the products in the models to overlap, as seen in Figure 9a. However the actual overlap can be seen in Figure 9b. We can see that only 12 products overlap between the ASU model and the cell-to-cell signaling model. There are an additional 78 products that the cell-to-cell signaling model misses.

To understand why there is such a small overlap we examine the features each model considers. The ASU team's experimental focus was on the interactions of protein features for the sender, regulator, and reporter, so the points of variation were chosen on these three variables. We expected a complete set of proteins to be documented on the cell-to-cell signaling page, but they are not comprehensively listed in the BioBrick repository.

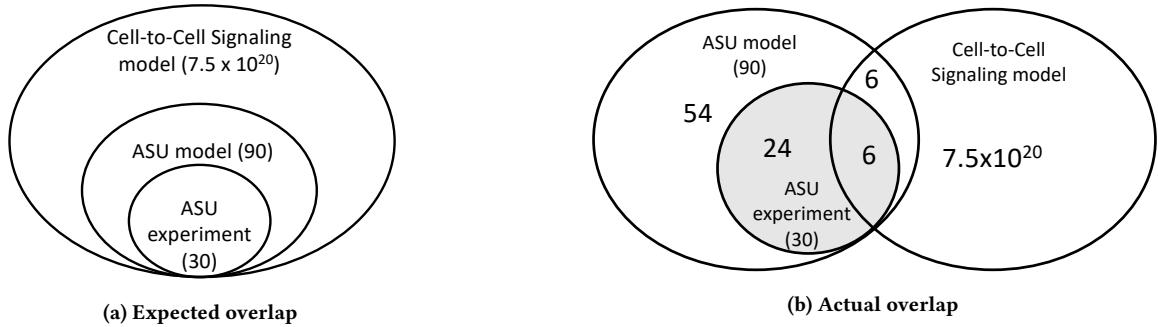
Both models are valid representations of quorum sensing systems, however they come from different resources and their products differ. This highlights a key problem with the current method of engineering a model - there is a lack of complete information available. In an ideal world we would show a complete list of all proteins, possibly through a central, open-source feature model.



**Figure 7:** The sender slice from a feature model for the 2017 Arizona State University’s iGEM project (ASU Model).



**Figure 8:** The receiver slice from a feature model for the 2017 Arizona State University’s iGEM project (ASU Model).

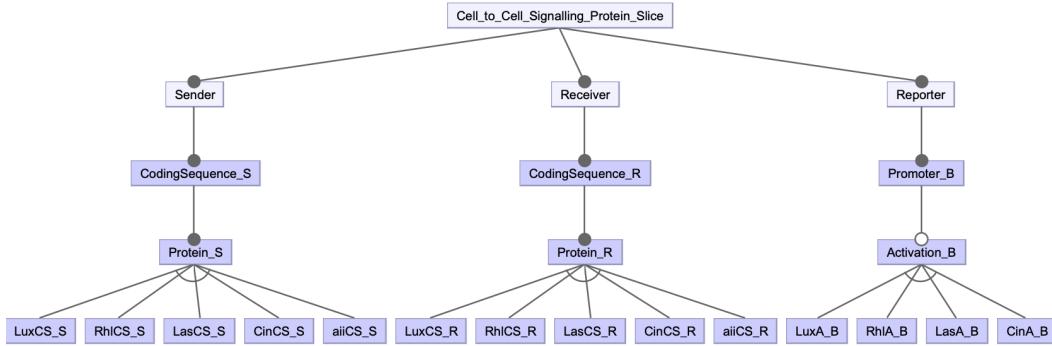


**Figure 9:** The overlap of the feature models and ASU experimentation. The sum of each circle is in parentheses.

**5.3.2 Testing and Analysis.** We next move to testing and analysis of our applications. Assume the ASU team used the cell-to-cell signaling model to drive their experiments instead of working from scratch. We demonstrate how they could have used this feature model to help with testing and analysis.

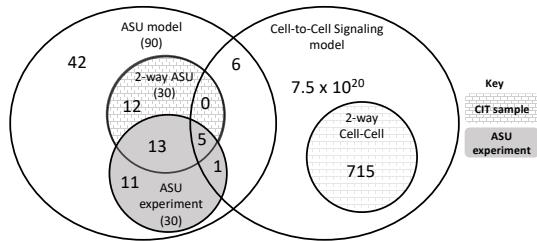
Since the ASU team focused only on the proteins, they could have *sliced* the cell-to-cell signaling model. This would yield the model in Figure 10 (called *protein slice*) which represents 100 products. This is significantly fewer than the total product space ( $7.50169 \times 10^{20}$ ), but more than what the ASU team eventually tested (30).

We could also employ common sampling methods such as combinatorial interaction testing (CIT) which samples broadly across a set of features [11]. Using sampling allows us to test a larger space of combinations. We used CASA [18] to build a 2-way CIT sample of the ASU model. In this scenario an interaction is between two proteins. We can cover all pairs of proteins in the complete model using 30 tests. Note that ASU also tested 30 products, but in order to scope the project they applied their own constraint that does not test for possible interactions between the regulator and reporter proteins. Using CIT will more broadly sample the interaction space of all three proteins.



**Figure 10: A slice of the complete cell-to-cell signaling feature model focuses on protein combinations to mimic the ASU team’s experiment (protein slice).**

We also built a 2-way CIT sample for the cell-to-cell signaling model, covering all pairs of *all features*. This has 715 tests. This is a significant reduction of the entire configuration space, however it may be too large in practice. Each of these samples and their product overlaps can be seen in Figure 11. All samples can be found on our associated website.



**Figure 11: Product overlap between each of the feature models and possible samples.**

**5.3.3 Leveraging Existing Reverse Engineering Tools.** Given that synthetic biology is an end-user software engineering domain, developers of organic programs may not be well versed in software development methods. Therefore, building a feature model from scratch may be a roadblock. We have observed this in other recent studies that have applied software engineering concepts to synthetic biology [16]. Instead, we want to understand whether it would be possible to obtain an initial feature model using only a set of products (which is how the ASU team built their experiments). This model could then be used to define CIT sampling, etc.

We utilized an existing reverse engineering tool, SPLRevO [34, 35]. We note that other similar tools could also be used. This tool accepts either (1) a set of constraints based on domain knowledge describing the compatibility of the DNA parts, or (2) a set of products which can be the known working composite components. The tool then uses a genetic algorithm to automatically build a feature model that represents all products. The fitness function (validity) aims to maximize the set of existing products while minimizing any additional products using a penalty. Because we do not have a set of constraints for this model we used a set of products as inputs. We

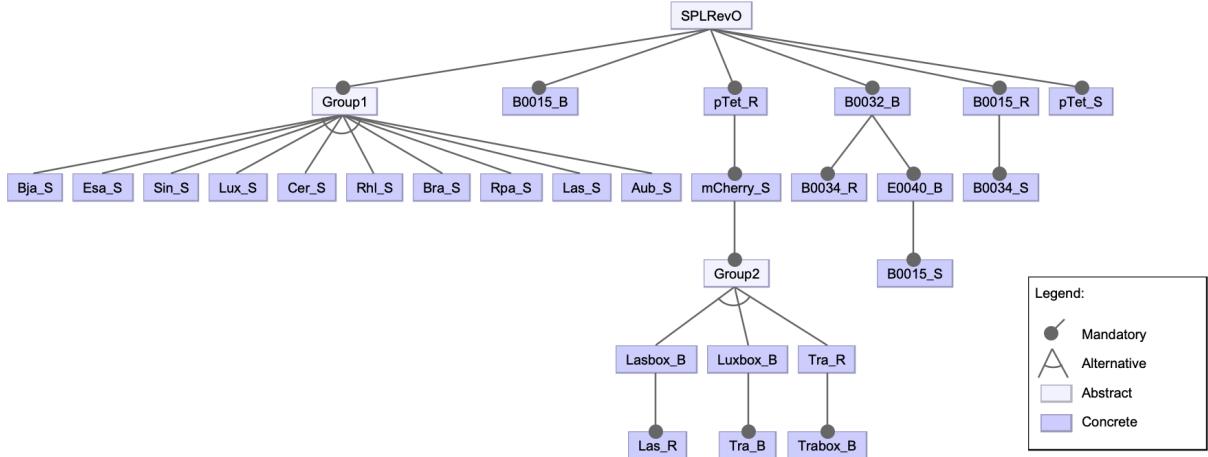
believe that a domain-specific language for users that describes the product line and which can generate constraints is an interesting avenue for future work.

The current prototype of SPLRevO can handle up to 27 features when reverse engineering from products. Therefore, we reduced the ASU model from 30 to 27 features by selecting two of the common features (B0010 and B0012) and merging them into one feature (B0015) for the sender, regulator, and reporter (e.g. B0010\_S+B0012\_S→B0015\_S, B0010\_R+B0012\_R→B0015\_R, etc.). Since there are 15 features in the ASU model that are common to all products, we could have chosen any of those features to combine or remove while still representing the same 30 products. SPLRevO returned the feature model seen in Figure 12. It was able to provide us with a model that closely resembles the hand-built model and has 100% validity (it represents exactly the same number of products). Though the models represent the same products, their physical structure is different. The SPLRevO model grouped the sender’s protein coding sequences together like the ASU model (Group1). Group2 represents the proteins for the regulator and reporter. Instead of adding a cross-tree constraint like the ASU team did for their experiment, the SPLRevO model uses mandatory relations for these under Group2. The rest of the features are all mandatory.

During this process we realized that 100% validity, while a good result for this study, might be relaxed in a real scenario. If the user provides a set of products, but they are not complete, relaxation could be used to allow them to interactively improve their model. We may also want to add additional features (from the BioBricks repository) and show the potentially larger model. Ultimately, if we can represent BioBricks features using a constraint language, there is an opportunity for direct reverse-engineering and greater scalability. SPLRevO has been evaluated on reverse-engineering a feature model up to 100 features when using input constraints [35].

## 6 DISCUSSION

We found several interesting insights when working on this study. First, we found the domain engineering from an open source repository like this to be challenging. We did not obtain the same feature model for cell-to-cell signaling as ASU, and cannot be certain whether they had additional domain expertise to restrict their study



**Figure 12: A reverse-engineered feature model using SPLRevO.**

or if this was due to chance. However, we believe that an interactive environment that we have described would be useful. Recent work on other SPL development in open source repositories are looking at ways to provide branching constructs for more easy reuse and understandability [25]. Second, there is an opportunity to develop more techniques for domain experts (who may not understand feature modeling) to work with product line engineers to build models that are functionally useful. It would be useful to provide some domain specific tools that can be used to generate sets of constraints, rather than require the user to either build the full feature model or list a set of products by hand. Third, we believe there is an opportunity for users to provide qualitative/experimental information which can be returned to the product line and connected via trace links. These can be provided as assets in the BioBricks Repository. Last, we note that there is a lot of noise in this repository. For instance, some parts do not have a DNA sequence which means it is empty code. We did not attempt to account for all of that during our study, but leave that as future work.

## 7 RELATED WORK

This paper follows a long line of research on software product lines. We do not attempt to summarize all of that work here, but point readers to several good surveys on this topic [3, 36]. Product line engineering has been applied in many emerging domains recently including drones and nanodevices [9, 24]. There is also a push towards open-source product lines [23, 25, 31] and the concept of a software ecosystem, where the community modifies and customizes product lines using a common platform and look [27].

The closest work is that of Lutz et al. [24] who study a family of DNA nanodevices. While they also look at DNA, they study chemical reaction networks (CRNs), rather than a living synthetically engineered organism. Their line of organic programming leverages CRNs, which are sets of concurrent equations that can be compiled into single strands of DNA [41]. CRNs have been shown to be Turing complete [14]. CRNs are not necessarily part of living organisms; there is no transcription and translation of the code [15].

Ours is not the first analysis of the BioBricks repository. Valverde *et al.* examined the relationships within the repository from a network perspective to gain an understanding of the software complexity (they too treat this as a software ecosystem) [37].

This work differs from the existing work in that we demonstrate the use of domain engineering to build a family of synthetic biology products which can be analyzed and reasoned about using traditional SPL engineering techniques.

## 8 CONCLUSIONS AND FUTURE WORK

In this paper we have shown how the emerging programming field of synthetic biology can potentially benefit from software product line engineering. We first presented the notion of an organic software product line. We then used the largest open source DNA repository to analyze 1) whether there are assets which are reused and products that share common and variable elements, 2) whether we can build feature models to represent the products in this repository, and 3) how common SPL techniques can be used to benefit product line development in this domain. We found reusable assets, commonality, and variability in the repository. We were able to build feature models to represent several common functions. We then demonstrated how we might automatically reverse engineer a model and how this can help users test and reason about the product space more comprehensively.

In future work we plan to investigate building a domain specific language for generating SPL constraints, and evaluating this approach in practice with teams of synthetic biologists perhaps in the iGEM Competition. We also plan to investigate challenges that are shared with modern SPLs including scalability, dynamic feature models, and collaborative SPLs.

## ACKNOWLEDGMENTS

We would like to thank the Haynes Lab at Emory University (formerly Arizona State University) for sharing additional artifacts with us. This work is supported in part by NSF Grant CCF-1901543, National Institute of Justice Grant 2016-R2-CX-0023, and by NSF Grant CBET-1805528.

## REFERENCES

- [1] James Anderson, Natalja Strelkowa, Guy-Bart Stan, Thomas Douglas, Julian Savulescu, Mauricio Barahona, and Antonis Papachristodoulou. 2012. Engineering and ethical perspectives in synthetic biology. *EMBO reports* 13, 7 (2012), 584–590. <https://dx.doi.org/10.1038/embor.2012.81>
- [2] Arizona State University 2017. ASU iGEM 2017: Engineering variable regulators for a quorum sensing toolbox. Retrieved June 13, 2019 from [https://2017.igem.org/Team:Arizona\\_State](https://2017.igem.org/Team:Arizona_State)
- [3] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: a literature review. *Information Systems* 35, 6 (2010), 615–636. <https://doi.org/10.1016/j.is.2010.01.001>
- [4] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-cortés. 2007. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceedings of the 1st International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS)*. 129–134.
- [5] Lara Tess Bereza-Malcolm, Gülay Mann, and Ashley Edwin Franks. 2014. Environmental sensing of heavy metals through whole cell microbial biosensors: a synthetic biology approach. *ACS Synthetic Biology* 4, 5 (2014), 535–546. <https://doi.org/10.1021/sb500286r>
- [6] James Bornholt, Randolph Lopez, Douglas M. Carmean, Luis Ceze, Georg Seelig, and Karin Strauss. 2016. A DNA-based archival storage system. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 637–649. <https://doi.acm.org/10.1145/2980024.2872397>
- [7] Yizhi Cai, Mandy L. Wilson, and Jean Peccoud. 2010. GenoCAD for iGEM: a grammatical approach to the design of standard-compliant constructs. *Nucleic Acids Research* 38, 8 (2010), 2637–2644. <https://doi.org/10.1093/nar/gkq086>
- [8] D. Ewen Cameron, Caleb J. Bashor, and James J. Collins. 2014. A brief history of synthetic biology. *Nature Reviews Microbiology* 12, 5 (2014), 381–390. <https://doi.org/10.1038/nrmicro3239>
- [9] Jane Cleland-Huang, Michael Vierhauser, and Sean Bayley. 2018. Dronology: An incubator for cyber-physical systems research. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE)*. 109–112. <https://doi.acm.org/10.1145/3183399.3183408>
- [10] Paul Clements and Linda Northrop. 2002. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA.
- [11] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. 1997. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 23, 7 (1997), 437–444. <https://doi.org/10.1109/32.605761>
- [12] Ramiz Daniel, Jacob R. Rubens, Rahul Sarapeshkar, and Timothy K. Lu. 2013. Synthetic analog computation in living cells. *Nature* 497, 7451 (2013), 619–623. <https://doi.org/10.1038/nature12148>
- [13] Michael B. Elowitz and Stanislas Leibler. 2000. A synthetic oscillatory network of transcriptional regulators. *Nature* 403 (2000), 335–338. Issue 6767. <https://doi.org/10.1038/35002125>
- [14] François Fages, Guillaume Le Guludec, Olivier Bournez, and Amaury Pouly. 2017. Strong Turing completeness of continuous chemical reaction networks and compilation of mixed analog-digital programs. In *Proceedings of the 15th International Conference on Computational Methods in Systems Biology (CMSB)*. 108–127. [https://doi.org/10.1007/978-3-319-67471-1\\_7](https://doi.org/10.1007/978-3-319-67471-1_7)
- [15] Martin Feinberg. 1987. Chemical reaction network structure and the stability of complex isothermal reactors—I. The deficiency zero and deficiency one theorems. *Chemical Engineering Science* 42 (1987), 2229–2268. [https://doi.org/10.1016/0009-2509\(87\)80099-4](https://doi.org/10.1016/0009-2509(87)80099-4)
- [16] Justin Firestone and Myra B. Cohen. 2018. The assurance recipe: facilitating assurance patterns. In *Proceedings of the International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, ASSURE Workshop. 22–30. [https://doi.org/10.1007/978-3-319-99229-7\\_3](https://doi.org/10.1007/978-3-319-99229-7_3)
- [17] Timothy S. Gardner, Charles R. Cantor, and James J. Collins. 2000. Construction of a genetic toggle switch in Escherichia coli. *Nature* 402 (2000), 339–342. Issue 6767. <https://doi.org/10.1038/35002131>
- [18] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering* 16, 1 (2011), 61–102. <https://doi.org/10.1007/s10664-010-9135-7>
- [19] iGEM API 2018. Registry of Standard Biological Parts API. iGEM Foundation. Retrieved June 13, 2019 from [https://parts.igem.org/Registry\\_API](https://parts.igem.org/Registry_API)
- [20] iGEM Competition 2018. International Genetically Engineered Machine Competition. iGEM Foundation. Retrieved June 13, 2019 from <https://igem.org>
- [21] iGEM Registry 2018. Registry of Standard Biological Parts. iGEM Foundation. Retrieved June 13, 2019 from <https://parts.igem.org>
- [22] Zoltán Kis, Hugo Sant'Ana Pereira, Takayuki Homma, Ryan M. Pedrigi, and Rob Kram. 2015. Mammalian synthetic biology: emerging medical applications. *Journal of The Royal Society Interface* 12, 106 (2015), 1–18. <https://doi.org/10.1098/rsif.2014.1000>
- [23] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux kernel variability model. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC)*. 136–150. [https://doi.org/10.1007/978-3-642-15579-6\\_10](https://doi.org/10.1007/978-3-642-15579-6_10)
- [24] Robyn R. Lutz, Jack H. Lutz, James I. Lathrop, Titus H. Klinge, Divita Mathur, Donald M. Stull, Taylor G. Bergquist, and Eric R. Henderson. 2012. Requirements analysis for a product family of DNA nanodevices. In *Proceedings of the 20th IEEE International Requirements Engineering Conference (RE)*. 211–220. <https://doi.org/10.1109/RE.2012.6345806>
- [25] Leticia Montalvillo and Oscar Diaz. 2015. Tuning GitHub for SPL development: branching models & repository operations for product engineers. In *Proceedings of the 19th International Conference on Software Product Line (SPLC)*. 111–120. <https://doi.acm.org/10.1145/2791060.2791083>
- [26] Alec A.K. Nielsen, Bryan S. Der, Jonghyeon Shin, Prashant Vaidyanathan, Vanya Paralanov, Elizabeth A. Strychalski, David Ross, Douglas Densmore, and Christopher A. Voigt. 2016. Genetic circuit design automation. *Science* 352, 6281 (2016). <https://doi.org/10.1126/science.aac7341>
- [27] Konstantinos Plakidas, Srdjan Stevanovic, Daniel Schall, Tudor B. Ionescu, and Uwe Zdun. 2016. How do software ecosystems evolve? A quantitative assessment of the R ecosystem. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC)*. 89–98. <https://doi.acm.org/10.1145/2934466.2934488>
- [28] Jiayuan Quan and Jingdong Tian. 2009. Circular polymerase extension cloning of complex gene libraries and pathways. *PloS one* 4, 7 (2009), 1–6. <https://doi.org/10.1371/journal.pone.0006441>
- [29] Ricardo A. Rosello and David H. Kohn. 2010. Cell communication and tissue engineering. *Communicative & Integrative Biology* 3, 1 (2010), 53–56. <https://doi.org/10.4161/cib.3.1.9863>
- [30] SBOL 2019. Synthetic Biology Open Language. SBOL Research Group. Retrieved June 13, 2019 from <https://sbolstandard.org>
- [31] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preibkschat, and Olaf Spinczyk. 2007. Is the Linux kernel a software product line?. In *Proceedings of the 2nd SPLC Workshop on Open Source Software and Product Lines*. 1–4.
- [32] Federico Tavella, Alberto Giaretta, Triona Marie Dooley-Cullinane, Mauro Conti, Lee Coffey, and Sasitharan Balasubramaniam. 2018. DNA molecular storage system: Transferring digitally encoded information through bacterial nanonetworks. (2018). arXiv:1801.04774
- [33] Stefan J. Tekel, Christina L. Smith, Briana Lopez, Amber Mani, Christopher Connot, Xylaan Livingstone, and Karmella Ann Haynes. 2019. Engineered orthogonal quorum sensing systems for synthetic gene regulation in Escherichia coli. *Frontiers in Bioengineering and Biotechnology* 7, 80 (2019). <https://doi.org/10.1101/499681>
- [34] Thammasak Thianniwit and Myra B. Cohen. 2015. SPLRevO: Optimizing complex feature models in search based reverse engineering of software product lines. In *Proceedings of the 1st North American Search Based Software Engineering Symposium (NASBASE)*. 1–16.
- [35] Thammasak Thianniwit and Myra B. Cohen. 2016. Scaling up the fitness function for reverse engineering feature models. In *Symposium on Search-Based Software Engineering (SSBSE)*. 128–142. [https://doi.org/10.1007/978-3-319-47106-8\\_9](https://doi.org/10.1007/978-3-319-47106-8_9)
- [36] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)* 47, 1, Article 6 (2014), 45 pages. <https://doi.acm.org/10.1145/2580950>
- [37] Sergi Valverde, Manuel Porcar, Juli Peretó, and Ricard V. Solé. 2016. The software crisis of synthetic biology. *bioRxiv* (2016). <https://doi.org/10.1101/041640>
- [38] Wilfried Weber and Martin Fussenegger. 2012. Emerging biomedical applications of synthetic biology. *Nature Reviews Genetics* 13, 1 (2012), 21–35. <https://doi.org/10.1038/nrg3094>
- [39] Wilfried Weber, Jörg Stelling, Markus Rimann, Bettina Keller, Marie Daoud-El Baba, Cornelia C. Weber, Dominique Aubel, and Martin Fussenegger. 2007. A synthetic time-delay circuit in mammalian cells and mice. *Proceedings of the National Academy of Sciences of the United States of America* 104, 8 (2007), 2643–2648. <https://doi.org/10.1073/pnas.0606398104>
- [40] William B. Whitaker, Nicholas R. Sandoval, Robert K. Bennett, Alan G. Fast, and Eleftherios T. Papoutsakis. 2015. Synthetic methylotrophy: engineering the production of biofuels and chemicals based on the biology of aerobic methanol utilization. *Current Opinion in Biotechnology* 33 (2015), 165–175. <https://doi.org/10.1016/j.copbio.2015.01.007>
- [41] Erik Winfree. 1995. On the computational power of DNA annealing and ligation. In *DNA Based Computers*. <https://resolver.caltech.edu/CaltechAUTHORS:20111024-133436564>
- [42] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. 2014. Patterns of folder use and project popularity: A case study of GitHub repositories. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, Article 30, 4 pages. <https://doi.org/10.1145/2652524.2652564>

# Automated Search for Configurations of Convolutional Neural Network Architectures

Salah Ghamizi, Maxime Cordy, Mike Papadakis, Yves Le Traon

SnT, University of Luxembourg

email:{firstname.lastname}@uni.lu

## ABSTRACT

Convolutional Neural Networks (CNNs) are intensively used to solve a wide variety of complex problems. Although powerful, such systems require manual configuration and tuning. To this end, we view CNNs as configurable systems and propose an end-to-end framework that allows the configuration, evaluation and automated search for CNN architectures. Therefore, our contribution is three-fold. First, we model the variability of CNN architectures with a Feature Model (FM) that generalizes over existing architectures. Each valid configuration of the FM corresponds to a valid CNN model that can be built and trained. Second, we implement, on top of Tensorflow, an automated procedure to deploy, train and evaluate the performance of a configured model. Third, we propose a method to search for configurations and demonstrate that it leads to good CNN models. We evaluate our method by applying it on image classification tasks (MNIST, CIFAR-10) and show that, with limited amount of computation and training, our method can identify high-performing architectures (with high accuracy). We also demonstrate that we outperform existing state-of-the-art architectures handcrafted by ML researchers. Our FM and framework have been released to support replication and future research.

## CCS CONCEPTS

- Software and its engineering → Software product lines; • Computing methodologies → Neural networks.

## KEYWORDS

Feature model, configuration search, NAS, Neural Architecture Search, AutoML

### ACM Reference Format:

Salah Ghamizi, Maxime Cordy, Mike Papadakis, Yves Le Traon. 2019. Automated Search for Configurations of Convolutional Neural Network Architectures. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3336294.3336306>

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336306>

## 1 INTRODUCTION

Nowadays, Deep Learning (DL) systems are used in a wide variety of domains as decision-making tools. Given an input (e.g., radiograph image, front view of a car, etc.), they are able to *classify* this input in a specific category or *class* (e.g. presence or absence of cancer tumor, appropriate steering angle, etc.). Such solutions offer self-learning capabilities: when *trained* with large amounts of labeled data (i.e., inputs with their associated output class), they automatically learn classes and predict, with some confidence, the classes of unseen inputs. This makes DL systems easy to use (require limited human intervention), flexible (adapt to many use cases), and effective (achieve high accuracy).

A DL system often consists of a classification model displaying the form of a *Deep Neural Network* (DNN), which is a network of neurons that includes the *input layer* (which encodes the characteristics of the given input), the *output layer* (which represents the classes to which the input can be classified) and, in-between, multiple intermediate *hidden layers*. For example, to classify inputs with 10 characteristics into 4 classes, we can use a 3-layer DNN, which is composed of one hidden layer that is connected both to the 10 neuron input layer (one input neuron per characteristic of the input) and to 4 neuron output layer.

From the above example, one can easily see that DNNs have parameters to configure. Usually, the number of neurons in the input and output layers are determined by the problem instance, but the number and shape of hidden layers are choices to be made by engineers. By adding more layers, one can create DNNs capable of solving complex tasks with potentially higher accuracy, yet at an increased computation costs (when training the model) and with a higher risk of *overfitting* (i.e., being accurate on training data but failing to classify well unseen data). In practice, there exist multiple alternative layer shapes that can be used, which adds complexity to the configuration process.

To develop appropriate DNNs, engineers resort on intuition, experience, trial and error, or generic models (models already developed for another purpose). This process is usually tedious, non-systematic and potentially sub-optimal (the resulting architectures may not achieve the best performance). To deal with this issue, we propose an automated search-based method that identifies prominent DNN architectures for given problems. More precisely, we rely on variability modeling techniques to represent the configuration space of DNN architectures, and configuration search techniques [8, 9] to search for optimal architectures within this space.

We start by modeling the variability within DNN architectures in a Feature<sup>1</sup> Model (FM) [14] extended with attributes and feature cardinalities [2, 4, 5, 22], where each valid configuration (aka variant,

---

<sup>1</sup>The term “feature” refers to a variation point in a variability system, and should not be confused with the input space characteristics that are usually referred to in ML.

product) of our FM corresponds to a deployable DNN architecture. We show that sophisticated FMs are convenient to represent the variability of DNNs by demonstrating how to derive, from an FM, DNN architectures that solve image recognition problems, e.g., LeNet5[18] and Inception modules.

We then implement a mapping from our FM to Tensorflow and Keras, which are standard programming libraries for Neural networks. Thus, from any valid configuration, we can automatically generate an implementation of the corresponding architecture in these libraries. On top of the above framework, we apply search algorithms to generate configurations according to some strategies, e.g., maximizing configuration diversity. To do so, we use PLEDGE [10], a configuration generation tool, and link it with Tensorflow+Keras. All in all, our framework automatically selects, builds and evaluates the accuracy of candidate architectures on specified datasets.

We evaluate our method on two image classification problems, which involve two commonly used datasets (MNIST and CIFAR-10). We use the LeNet5 and SqueezeNet (state-of-the-art DNN architectures) as a baseline for comparison. Our results show the capability of our method to explore the configuration space and to search for good architectures. Notably, we manage to find architectures that outperform LeNet5 and SqueezeNet in raw accuracy on both datasets.

Overall, our work paves the way for applying software product line and configuration techniques to DL systems. We offer a practical and feasible way to automatically optimize DNNs by specializing their configurations on particular domains. We, thus, deemed our endeavor successful and expect future research to contribute with novel configuration search techniques tailored to this context. To support this task, we make our models and framework publicly available.

## 2 MOTIVATION AND RESEARCH QUESTIONS

Our goal is to provide engineers with a tool to configure and specialize (to specific tasks) DNNs. To achieve this, we rely on a variability model that captures all the variation points of DNN models together with the validity constraints. As typically performed in configurable systems, (valid) configurations selected from the variability model are linked to specific implementations (in our case, libraries such as Tensorflow and Keras) to derive deployable DNNs. Based on this framework, engineers can train and evaluate the resulting DNNs according to their needs.

We aim at representing the whole variability of DNNs with an FM. In this first endeavor, we focus on the architecture of DNNs (i.e. how their neurons are structured and connected) and leave aside the training process (training rate, cost function, gradient descent method, etc.) and the dataset preparation (e.g. data augmentation) since the architecture space is the most complex part (see Section 3 for more details). In view of this, our first research question is:

**RQ1:** Can we develop a variability model that represents all possible DNN architectures?

Having formed the variability model, we turn our attention to the configuration process. We aim at specializing DNNs to particular

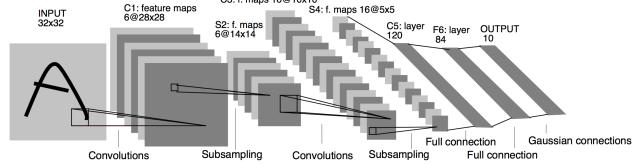


Figure 1: LeNet5 architecture proposed in the original paper

classification tasks by exploring and evaluating the performance of selected configurations. As the configuration space is large and constrained, we rely on techniques that have been successfully used to generate configurations of configurable systems. Our second research question is thus formed as:

**RQ2:** Can we effectively search the configuration space and identify well-performing DNN architectures?

Ultimately, the benefits of our configuration search are the identification of high-performance architectures. This means that specializing a DNN architecture to a particular task should lead to optimal or nearly-optimal results. We evaluate this by comparing the automatically generated architectures with state-of-the-art DNNs. Thus, we ask the following question:

**RQ3:** Does our technique finds DNN architectures that outperform the state of the art?

## 3 A VARIABILITY MODEL FOR DNNs

While DNNs are classically viewed as a linear sequence of fully-connected neuron layers (often named *dense layers*), new layer structures have been introduced over the past twenty years. For example, Convolutional Neural Networks (CNNs)[18] – mostly used in image recognition – have introduced the convolution layer and the pooling layer. The former produces a set of patterns generated by convolving successive filters on an input signal [17], thereby emulating the behavior of the visual cortex. To do so, it applies matrix operations to merge multiple sets of information. On the other hand, a pooling layer aggregates an input matrix into a smaller one to reduce computation cost and overfitting.

One of the most established DNN architectures based on convolution layers is LeNet5, of which Figure 1 depicts the structure as presented originally by LeCun et al. [18]. Its input is a matrix (usually of pixels) which is analyzed through a three-step process. Each of the first two steps involves a convolution layer and a pooling layer (named subsampling in Figure 1). The last step consists of going through one convolution layer and one dense (fully-connected) layer that, in the end, performs the classification itself.

More recently, Inception Network [31] architectures popularised new constructs like branching and merging of layers, while Residual Network (ResNet) [7] architectures can make neurons' output connections skip the next layers to propagate their computation farther in the network. All those advanced mechanisms make the classical, over-simplistic way of seeing DNNs inappropriate to represent the variability of their architectures.

This raises the need for a generalized representation allowing the combination of fine-grained constituents of any of the above layer structures. We believe that FMs are an appropriate modeling formalism to this end. Indeed, their compositional form allows the definition of a hierarchical structure where concepts are further and further decomposed while providing the ability to assemble a holistic system from those inner constituents. Moreover, since their introduction by Kang et al. [14], they were extended over the years to model the variability of an increasing variety of systems.

Figure 2 shows our FM of DNN architectures designed using FeatureIDE [33]. An architecture consists of two mandatory features representing the Input structure (corresponding to the initial matrix input to the DNN) and the Output structure (corresponding to a dense layer with a softmax activation function and whose number of neurons is the number of classes), as well as a multi-feature Block.<sup>2</sup> The instances of Block together represent the hidden layers of the DNN, lying between the input layer and the output layer. Each instance includes one child multi-feature named Cell, which represents the particular structure we invented and that generalizes over the traditional structure of neuron layers.

The structure of a cell is better illustrated in Figure 3. It is constituted by five elements, each of which corresponds to a child feature of Cell in the FM. Input 1 and Input 2 both represent a layer (named an input layer of the cell) that receives an input (i.e. a matrix) to the cell and reason on it. Input 2 is in some sense optional, as it can be filled with zeros to simulate that the cell accepts one input. Operation 1 and Operation 2 are operations applied to the results of the two input layers in order to prepare their combination. Combination defines what type of combination is performed. Finally, Output defines how far in the network the result of the combination must be sent.

Before detailing each of those constituents, we first give an example of an architecture modeled as blocks and cells. In Figure 4, we illustrate the use of our cell-based representation to model the LeNet5 architecture previously introduced in Figure 1. Each of the three blocks corresponds to one of the three steps performed by a LeNet5 DNN. In each of the first two blocks, we find two cells: one corresponding to the convolution layer and the other one to the pooling layer. The third block has two cells: one for the convolution layer and the other for the dense layer. Since in LeNet5, the input simply goes from one layer to another, all cells use zeros as the second input layer, *void* operations (i.e. no operation), and combine results by summing them.

**Inputs.** In addition to Zeros layer, input layers can be of four types, represented by alternative features in Figure 2. Identity simply keeps the input of the cell as it is. Dense corresponds to a layer of fully-connected neurons. It has two attributes: *neuron\_number* designates the number of neuron in the layer, while *activation* denotes their activation function (e.g. sigmoid, softmax).

Pooling aggregates meaningful information from the cell's input. It operates on a matrix and can either extract the average or the maximum of every square area of that matrix. The size of the areas is defined by its attribute *kernel1*, while the *type* attribute defines whether to extract the average or the maximum. Pooling layers can also skip elements of the original matrix and the amount

of skipped elements after each extraction is defined by the *stride* attribute. Last, the attribute *padding* defines how to deal with parts of the extracted area that are out of the original matrix, i.e. filling the missing parts with zeros or discarding the whole area.

Convolution corresponds to a convolution layer and has multiple attributes. *Kernel1* is the size of the filters that will be used, where a filter is a matrix that allows the convolution to detect different patterns (primitive patterns for images include, e.g., edges or corners). Additionally, the *filters\_number* attribute denotes how many filters this matrix will try to learn. The *stride* and *padding* attributes are defined as in Pooling inputs, while *activation* denotes the activation function used by the layer.

**Operations.** The most common operation is *Void*, meaning that it does not alter the result of the input layer. Another is *Flatten*, which converts the result matrix into a vector. *Padding* adds zeros to the matrix to change its size, while *Activation* denotes a specific activation function used to modify the value boundaries of matrix elements. Some operations can also be used to introduce regularization like *Dropout* – which simulates the random dropping of neurons with a probability rate given by the *feature* attribute *rate* – or *BatchNormalization*, which normalizes the activations of the input layer at each batch.

**Combination.** There are three types of combination: *Sum* simply sums the results, *Concat* concatenates them, while *Product* performs a dot product matrix operation.

**Output.** We distinguish between three types of destination to which a cell's output is sent. It can be a subsequent cell of the same block (i.e. *Cell output*) or the next block (i.e. *Block output*), or the output is one of the final outputs of the network (i.e. *Out output*). A *Cell output* has an attribute named *relativeIndex*, which indicates that the output is transmitted to the (*relativeIndex*+1)th next cell.

This description of the constituents of a cell completes our definition of the FM's features. In addition to the feature decomposition groups, cross-cutting constraints define validity rules for the architecture. Due to lack of space, we do not represent them as logic formulae but, instead, we explain them intuitively. First, the *i*-th block/cell can only exist if the (*i*−1)-th exists as well. Besides, if the output of a cell is a *Block output*, there must exist a following block. If it is a *Cell output*, there must be at least a number of subsequent cells equal to *relativeIndex* + 1. Relative indexes must also be defined such that a cell is not the target of more than two outputs. Additionally, Pooling and Convolution layers are only executed on matrices whose size is compatible with their kernel size. Flatten operations cannot be applied on 1-dimensional matrices. Combinations (sum, concatenation, and product) must be performed on two matrices with compatible size.

**LIMITATIONS.** Overall, our FM encompasses a large class of feedforward neural networks. It does not, however, support Recurrent Neural Networks (RNNs), which would require looping connections between neurons. This restriction is due to the linear structure of our blocks and cells. Extending our modeling to support RNNs is feasible, but would require (i) allowing cells' output to reach a preceding cell or block and (ii) introduce new input layers, operations, and combination.

<sup>2</sup>A multi-feature is a feature with a maximum cardinality greater than one.

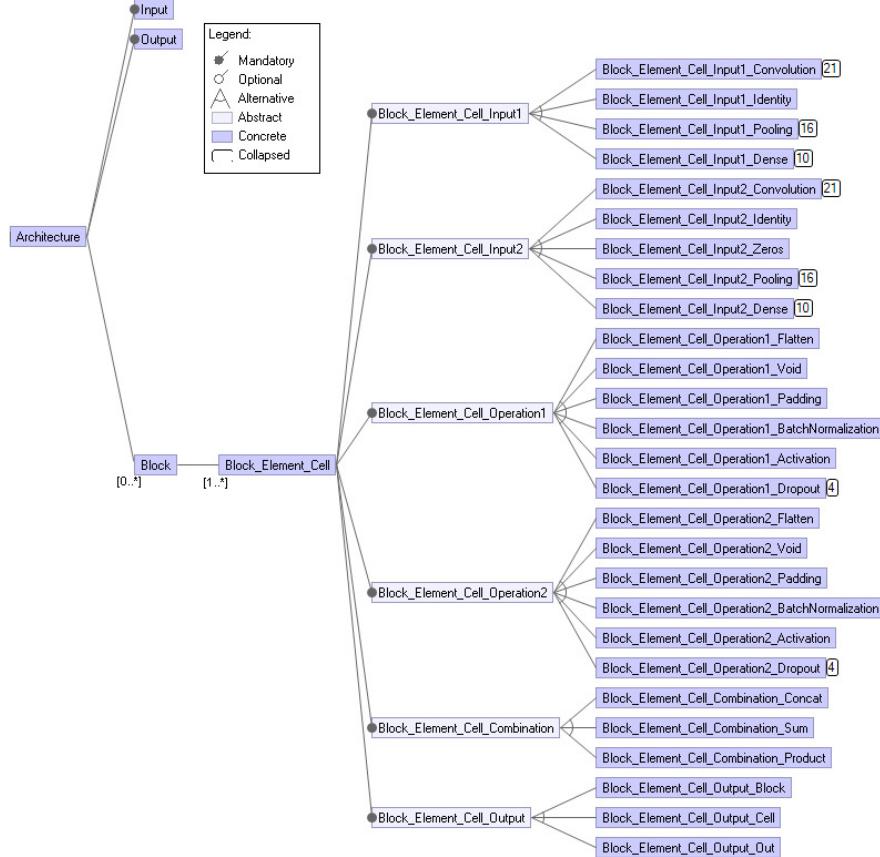


Figure 2: FM of DNN architectures, with feature attributes collapsed. The FM has been built using FeatureIDE [32].

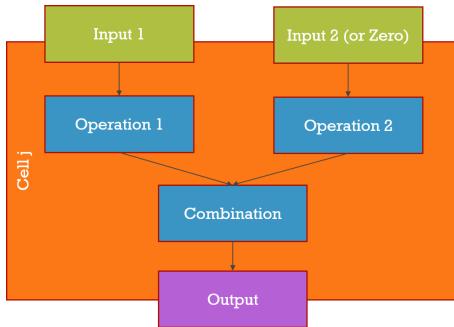


Figure 3: The inner constituents of a cell.

## 4 IMPLEMENTATION

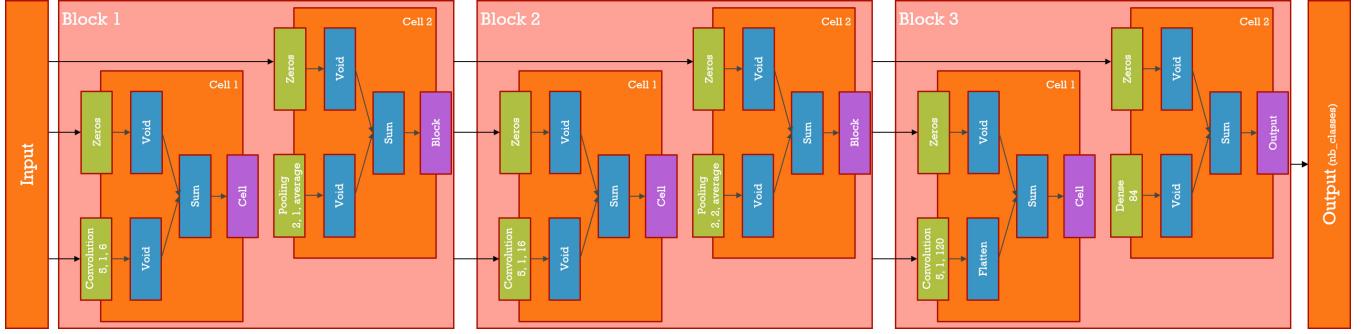
### 4.1 Forming Architectures

Many automated techniques have been proposed to generate/select configurations of variability-intensive systems. Some of them attempt to cover all (1-wise, pair-wise, and more generally t-wise) feature interactions [12]. In this work, we use diversity (in terms of enabled/disabled features of the FM) [9, 10] as a driving criterion. Intuitively, the generation method should allow the formulation of

architectures with diverse shapes (in terms of layers). Such techniques have been intensively studied and compared in the area of configurable systems [21, 34]. Here, we aim at demonstrating that it is possible to represent and search the DNN architecture space so that we can find useful configurations. Therefore, we choose diversity as it has been shown to be effective at global space exploration [6, 9].

More precisely, we use the PLEDGE tool [10], which generates a given number of configurations by using a search algorithm that maximizes the diversity of the provided sample. Here, the term diversity refers to the mean distance (on the feature space) between any two configurations of a given sample. By feeding our FM into PLEDGE, we can thus sample a diverse set of DNN architectures.

The drawback of PLEDGE is that it does not handle feature cardinalities and feature attributes. Therefore, we rely on the array-based semantics of extended FMs proposed by Cordy et al. [4] to (i) transform numeric and enumerated attributes into alternative Boolean features, (ii) unfold each multi-feature in a corresponding sequence of single features with the same underlying structure, (iii) transform all constraints accordingly. This process resulted in an increased size of our FM which, e.g., includes 3,296 features and 8,004 constraints when both the numbers of blocks and cells per block are limited to 5.



**Figure 4: A LeNet5 architecture modeled using our representation. It contains three blocks where the first two are Convolution-Pooling steps and the last block operates the reasoning with Dense Layers**

## 4.2 Deploying DNN architectures

We have developed a prototype tool (named *FeatureNetCompiler*<sup>3</sup>) on top of the Tensorflow and Keras frameworks. It takes as inputs configurations (selected by PLEDGE) and generates the corresponding DNN models in Tensorflow+Keras. Those models are obtained by building the DNN architectures defined by the configurations and using default training parameters. Our tool then trains and tests the generated models on a given dataset.

A particular difficulty we faced when implementing our tool was the handling of cells' outputs that are not transmitted to the next cell. To achieve this, we designed a data structure to store the pending cell outputs. In this structure, each cell's output is stored together with a number equal to its *relativeIndex* attribute. Initially, this number represents the distance between the cell that has produced the output and the cell that will consume it. An output is said to be *active* when its associated number reaches zero. Then, a newly reached cell uses the active elements, or the inputs of the block if there are no such elements. As active elements of the heap are used, they are removed from the data structure and the associated number of all remaining elements is reduced by one. This mechanism allows the merging of multiple chains into one, and supports complex branching, skipping and merging, which is required to model, e.g., an Inception Module [31].

## 5 EVALUATION PROTOCOL

### 5.1 Methodology

RQ1 aims at evaluating the conformance of the FM we presented in Section 3. To answer this RQ, we demonstrate that we can form existing and arbitrary deployable architectures. In particular, we check whether we can form architectures from 1,000 configurations produced by PLEDGE and two manually-configured architecture (LeNet5 [18] and Inception [31]). We thus check whether we can deploy and train these architectures. To further validate our model, we also check the semantic equivalence between the LeNet5 architecture generated by our tool and the manually-built counterpart of LeNet5 (implemented directly in Tensorflow/Keras). This is checked by comparing the average accuracy over multiple runs (on both training and test sets) during the model evolution (i.e. the training

epochs). Note that we do not check semantic equivalence on Inception, as it would require excessive computing power (multiple weeks to train it with the same experimental protocol).

RQ2 concerns the capability of our approach to search for well-performing architectures. To evaluate this, we form architectures based on 1,000 configurations produced by PLEDGE. We train these architectures for 12 epochs, on the selected datasets, and compute their accuracy. Since the configurations are diverse, we expect to see a wide range of accuracy values.

RQ3 compares our architectures with the state of the art. To enable a fair comparison, we use LeNet5 (as parameterized in [18]) and SqueezeNet [11] as baselines. These two architectures have a size (expressed in a number of weights) of the same order of magnitude as the median size of architectures we generate. We compare the accuracy and the efficiency of all the architectures on the datasets, when trained for 12, 300 and 600 epochs. In case our generated architectures achieve better accuracy and efficiency than LeNet5 and SqueezeNet, this would mean that the cumbersome process of manually constructing efficient DNN architectures can be automated.

### 5.2 Experimental Setup

**Forming architectures.** Unsurprisingly, preliminary experiments revealed that training a large number of large-sized architectures requires tremendous computation resources. In order to keep the required computation time at a reasonable experimental level (less than 24 hours), we applied two restrictions. First, we limited the configurations generated by PLEDGE to 1,000. Second, we added the following constraints in our FM:

- The maximum number of blocks and the maximum number of cells per block are set to 5.
- Convolution layers include at most 128 filters, each of which has a maximum size of 5x5.
- Dense layers have at most 512 neurons each.
- Batch normalization and padding operations are forbidden.
- Concatenation and product combinations are forbidden.
- There can be only one `Out` output (more would typically be used to improve training).

The above constraints allow keeping training time under a reasonable threshold, even though some generated architectures still

<sup>3</sup>publicly available at <https://github.com/yamizi/FeatureNet>

take more than 2 hours when trained for 600 epochs. However, it prevents us to generate more costly architectures like ResNet [7], Inception, and AlexNet [17]. Still common and even some recent architectures remain available, in particular, LeNet5 and SqueezeNet, which we use as baselines in our experiments.

**Training setup.** In all our experiments, we rely on a commonly used setup for training. We apply stochastic gradient descent with categorical crossentropy as loss function, a learning rate of 0.01 without any decay or momentum, and a batch size of 128. We did not try to optimize these hyperparameters, as they are likely architecture-dependent and we wanted to focus only on sampling architectures and evaluating them on common ground.

**Datasets.** We consider two datasets of image recognition problems that are widely used in research on machine learning. The first dataset is *MNIST*, which is composed of 28x28 greyscale images, each of which represents one of the ten digits. We used 60,000 images for training and 10,000 images for testing. The second dataset concerns *CIFAR-10*, a collection of 60,000 labeled images scattered in 10 classes, including 50,000 for training and 10,000 for testing.

**Performance metrics.** Our experiments make use of two performance metrics that are commonly used to evaluate DNN models. The first one is the classical accuracy, which is defined as the ratio of images that are well-classified by the DNN model. A higher accuracy thus means more correct predictions. The second is the *efficiency* of the model, that is, the accuracy divided by the total number of weights computed in the model, which is an indication of the resources required to train it. Hence, efficiency represents a tradeoff between the correctness of predictions and computation resources.

**Hardware.** We run PLEDGE on a laptop with CoreI7-8750H processor and 16GB RAM. The training and testing of neural networks was performed on a Tesla V100-SXM2-16GB GPU.

## 6 RESULTS

### 6.1 RQ1: conformance of Feature Model

To check the conformance of our FM, we successfully deployed, trained and test (with the related datasets) 1,000 configurations generated by PLEDGE, as well as LeNet5 and Inception. This fact shows that our model leads to valid architectures.

To further check the validity of these architectures we semantically compare the LeNet5 architecture of our model (representation displayed in Figure 4) with the one implemented directly in Tensorflow/Keras (with the same parameters). Figure 5 shows the training and test accuracy on CIFAR-10 (averaged over 10 runs). We observe that both types of accuracy follow the same trend in both models over the epochs. This confirms the conformance of our implementation. Here it must be noted that the slight recorded differences are inherent to the stochastic nature of the training process.

Figure 6 shows an inception module modeled after our representation. This shows the capability of our modeling to support the branching and merging behavior of such modules. Overall, an inception module concatenates 4 layers at once. We achieve this by performing 3 concatenations (in Cell 5, Cell 8 and Cell 9). For instance, Cell 5 concatenates the outputs from a convolution with kernel size 3 (Cell 2) and a convolution with kernel size 5 (Cell 4).

All in all, our results show that our FM can represent the space of DNN architectures and can successfully automate the generation of their variants.

### 6.2 RQ2: Searching the configuration space.

Figure 7a shows the accuracy values obtained by our configurations when trained for 12 epochs. For each percentage of accuracy, we draw the number of architectures (out of the sample of 1,000) that achieve at most this accuracy percentage. For instance, we observe that 585/1000 architectures have an accuracy lower than 50%, while that number jumps to 852 for 90% accuracy. In the end, only 15 architectures (i.e. 1.5% of the sample) achieves more than 95% of accuracy.

Figure 7b shows the results for CIFAR-10. We observe that our configurations are scattered over different ranges of accuracy indicating large performance differences. CIFAR-10 being more challenging than MNIST, 729 configurations barely reach 30% of accuracy, while only 994 are below 50%. Overall, none of the architectures manage to achieve 55% of accuracy, due to our limitation to 12 epochs. Later in Section 6.3, we study how the best architectures obtained at 12 epochs perform when they are trained for more epochs.

Overall, our results confirm that the DNN architectures have a substantial impact on accuracy and that searching the configuration space leads to architectures that significantly outperform others. We also observe that architectures with high accuracy are not necessarily the ones with the largest size, while architectures performing poorly are found in all size ranges. This further supports the motivation for searching through the whole configuration space and not restricting it to large-sized architectures.

Taken together with the finding of RQ1, we can claim that our framework has the ability to form and specialize architectures for a particular domain, i.e., architectures that perform best in a particular domain.

### 6.3 RQ3: Finding architectures outperforming the state-of-the-art.

To conduct a fine-grained evaluation of our approach, we generate architectures three times under different constraints. The first sample, denoted S1, denotes the 1,000 architectures obtained by applying the general setup described previously, which thus cannot be substantially larger than LeNet5 as parameterized in [18]. The second one, denoted S2, restricts the sampling to 100 LeNet5 architectures with different parameterizations. Finally, the third one consists of 100 architectures and results from applying additional constraints to S2 that enforce smaller architectures: 5x5 convolution kernels are replaced by 3x3 kernels, convolution strides are increased from 1x1 to 3x3. In the following experiments, we retain the most accurate architectures obtained from the three samples, which we denote by *Top S1*, *Top S2*, and *Top S3*, respectively. Note that, in what follows, we use a slight abuse of language: by LeNet5, we refer to the LeNet5 architecture as it is parameterized in [18]. By contrast, S2 samples over the whole range of parameterizations of LeNet5 architectures.

**6.3.1 MNIST, 12 epochs.** The top part of Table 1 shows the accuracy, size (in a number of weights) and efficiency of LeNet5 [18] and our

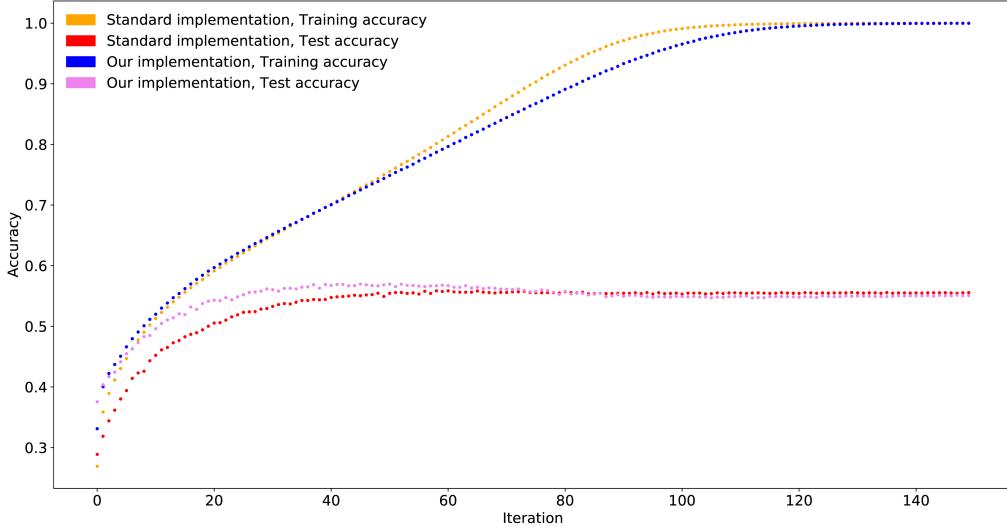


Figure 5: Comparison of the average accuracy of a standard LeNet5 implementation with the one we generate, averaged over 10 training runs.

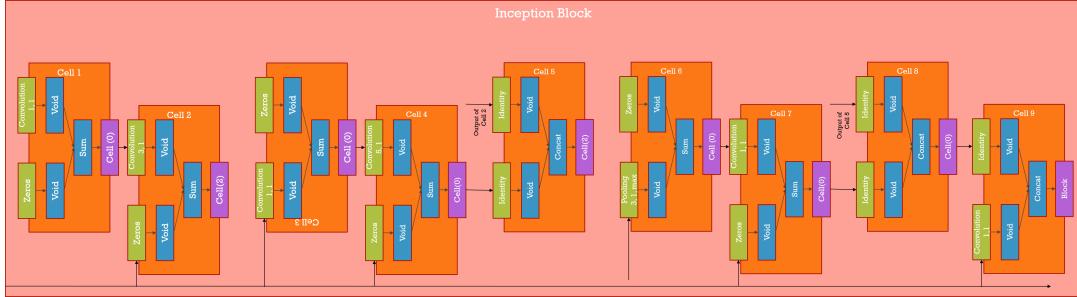


Figure 6: An inception module (part of inception architecture) modeled using our representation.

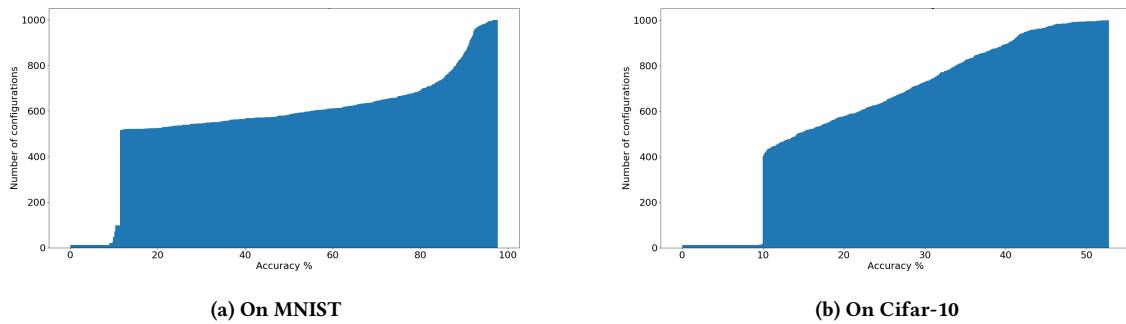


Figure 7: Distribution of the 1,000 generated architectures over all percentages of accuracy on two datasets. Any point  $(x, y)$  of the graph denotes that  $y$  architectures achieve an accuracy lower than  $x\%$ .

top sampled architectures, where  $efficiency = \frac{accuracy \times 10^6}{size}$ . We observe that, by selecting architectures of similar size to LeNet5, we obtain an architecture (*Top S1*) which performs better than LeNet5. It is actually slightly more accurate than LeNet5 while being 34% smaller, thereby leading to increased efficiency. When

we restrict the sampling to 100 LeNet5 architectures, the most accurate we obtained (*Top S2*) is, overall, slightly more accurate and slightly bigger than LeNet5. *Top S3* illustrates an interesting tradeoff between accuracy and size, as it is smaller than the other three by one order of magnitude, while still providing an accuracy of 92.31%.

**Table 1: Accuracy, size, and efficiency of LeNet5, SqueezeNet and our best sampled architectures, with a 12-epoch training, on MNIST (top part) and CIFAR-10 (bottom part).**

Dataset	Architecture	Accuracy	Size	Efficiency
<b>MNIST</b>	LeNet5	97.14%	545546	1.78
	Top S1	97.74%	365194	2.68
	Top S2	97.65%	570218	1.71
	Top S3	92.31%	43578	21.18
	SqueezeNet	43.67%	858154	0.51
<b>CIFAR-10</b>	LeNet5	49.13%	868406	0.57
	Top S1	52.77%	2494858	0.21
	Top S2	57.79%	862646	0.67
	Top S3	37.44%	38842	9.64
	SqueezeNet	17.96%	876970	0.51

This leads to efficiency 12 times higher than LeNet5. We also observe that SqueezeNet performs poorly on MNIST. We see two reasons behind this. First, like most recent architectures, SqueezeNet is tailored to handle more challenging datasets (with larger and more detailed images) than MNIST and CIFAR-10. Second, it may require more training epochs to reach an acceptable accuracy.

**6.3.2 CIFAR-10, 12 epochs.** Next, we repeat the same experiments on the CIFAR-10 dataset. We also consider the SqueezeNet architecture, which is known to offer an excellent efficiency on expensive datasets [11]. Results are shown in the bottom part of Table 1. While *Top S1* obtains higher accuracy than LeNet5, this comes at the cost of a tripled size, thereby leading to a worst efficiency. However, by restricting the generation to LeNet5 architectures, we obtain an architecture (*Top S2*) of similar size that offers a substantial increase of accuracy (from 49.13% to 57.79%). The architectures resulting from the third sample reach 37.44% of accuracy at best, but exhibit size more than 20 times smaller than LeNet5, which leads to a way higher efficiency. Like previously, SqueezeNet performs poorly on CIFAR-10. According to its inventors [11], this architecture can achieve more than 90% accuracy on CIFAR-10 by tweaking the training process (with momentum and decay), augmenting the data and applying various regularization methods. Such improvements are, however, out of the scope of our study as we are only focusing on the architecture structure and its parameters.

Overall, we conclude from both experiments that, with a 12-epoch training, our approach indeed manages to outperform an established architecture that was designed manually, be it on accuracy or efficiency, and that effective architectures are not necessarily the largest.

**6.3.3 CIFAR-10, 600 epochs.** To confirm that the above results are not due to the limited amount of epochs we allowed for training, we conducted additional experiments over more epochs. More precisely, we retain the 10 architectures resulting from the first generation S1 that have the highest accuracy. Then, we train them for a total of 600 epochs on the CIFAR-10 datasets and compare the obtained performance metrics against LeNet5 and SqueezeNet. We limited the total training of each architecture to 100 minutes; only two of our architectures ended their training prematurely because of that threshold.

**Table 2: Accuracy of the 10 best architectures from S1, LeNet5 and SqueezeNet on CIFAR-10 and at 12, 300 and 600 training epochs. Architectures are ordered by descending order of accuracy at 600 epochs. \* indicates shortened training.**

Architecture	Size	(12)	(300)	(600)
#063	0.45M	48.25%	<b>74.28%</b>	<b>74.74%</b>
#203	0.17M	52.64%	64.55%	65.25%
#161	3.62M	48.57%	64.46%	65.24%
#477	2.49M	<b>52.77%</b>	63.43%	64.25%
#444	15.73M	49.97%	62.40%	62.80%
#143	12.68M	51.37%	60.46%	60.17%
LeNet5	0.87M	49.13%	59.42%	59.26%
#634	1.43M	51.11%	59.76%	* 59.06%
#936	0.04M	48.92%	54.26%	56.33%
#595	134.22M	52.16%	52.64%	* 53.64%
SqueezeNet	0.88M	17.96%	46.17%	49.69%
#059	31.51M	51.9%	52.10%	49.47%

Table 2 shows the results at 12, 300 and 600 epochs, where the examined architectures are ordered by descending order of accuracy at 600 epochs. We observe that the gap between the best-performing architecture and LeNet5 tends to increase as more epochs are allowed for training (74.74% against 59.26%), which confirms the effectiveness of our approach to discover better architectures. Although the accuracy of SqueezeNet increases drastically over the epochs, it remains lower than most of the others, including LeNet5. Interestingly, the architecture performing the best at 300 and 600 epochs was the worst one (if we except SqueezeNet) at 12 epochs.

We also see that, while the accuracy of all architectures increases from 12 to 300 epochs, four of the ten sampled ones have a slightly lower accuracy at 600 epochs than at 300 epochs, likely due to overfitting on the training set. To investigate this further, we show in Figure 8 the evolution of the training accuracy (i.e. accuracy obtained by evaluating the architecture on the training set) and of the test accuracy (i.e. on the test set) for all the twelve architectures and from 1 to 300 epochs.

First, these results confirm that architecture size is not the main factor for accuracy and its evolution over the epochs. For example, #143 and #444 have comparable sizes but behave very differently, with #143 overfitting more and more on the training data. Furthermore, the architectures where the gap between training and testing accuracy is the biggest are not necessarily the largest architectures.

Architectures #063 and #444 are the ones where training accuracy and test accuracy remain close and keep on increasing over the epochs. This is a good indicator that they can achieve even higher accuracy with more training. It also explains the fact that #063 was relatively bad at 12 epochs while it becomes the best afterward. It also means that some architectures from our initial 1,000 generated set that we discarded at 12 epochs could end up outperforming the final ten in the long run.

Conversely, other architectures, such as #059 and #936, see their test accuracies increase until approximately 50 epochs before it decreases. If we look closely at LeNet5, we observe that it rapidly

overfits on the training set as well. As a consequence, its test accuracy tends to stabilize at 50 epochs and does not seem to progress further.

## 6.4 Threats to Validity

Validity threats may arise due to our implementation. Our FM model might not be entirely accurate and our implementation might not be entirely correct. We do not consider this threat as major since we checked our models and implementations using a large sample of configurations. Moreover, our technique relies on widely-used frameworks, Tensorflow and Keras, and publicly available architectures (LeNet-5 and SqueezeNet) as a comparison baseline, all of which have stood to expert scrutiny. Overall, to reduce these threats we make our models and implementation publicly available.

Another threat regards the accuracy metrics we use on the two image datasets, MNIST and CIFAR-10. While accuracy is the commonly used metric, it does not reflect the extent to which our data architectures overfit. As we did not experiment with datasets involving a large variety of many or few classes and/or bigger images, it could be the case that our results are coincidental and do not generalize to other cases. Though, our framework manages to identify architectures leading to a large range of results indicating its ability to specialize in specific cases.

Other threats may affect the generality of our framework. We exclusively focused on DNN architectures and eliminated all optimizations of the training process, such as data augmentation[30], modern regularization techniques (e.g. group-lasso regularizer [36], global average pooling, [19], etc.), learning rate decays and improved stochastic gradient descent methods [28] (e.g. Adam [16]) in order to perform the experiment in a reasonable amount of time. All these innovations may impact both the accuracy and the generalization (to unseen data) of DNNs. By adding such mechanisms to our study, we can likely get better accuracy for both our generated models and the baseline ones.

Another threat may arise due to our restrictions, i.e., we constraint the size of the architectures due to our limited computation resources. More powerful hardware and longer training times would be necessary to scout larger search spaces and more complex configurations. Still, our study already shows promising results on smaller search space and could be extended in the future to corroborate our conclusions on larger architectures.

Finally, we compared our architectures with two references architectures, LeNet5 and SqueezeNet. Still, we consider them as representative because they form the current state-of-the-art. These architectures require a similar amount of weights to train on Cifar-10, about 0.88 million, which is of the same scale as our best architecture (#064). Besides, while LeNet5 is an old architecture that only uses the basic concepts of Convolution and Pooling, SqueezeNet which was proposed in 2017 uses a large variety of innovations, such as ReLU activations, Dropout, Skip connections, Convolution filters concatenation and a wide range of kernel sizes and number of features in each of its layers. SqueezeNet is, therefore, a good summary of the modern techniques proposed to improve image classification without massively increasing the size of the neural network.

## 7 RELATED WORK

### 7.1 Neural Architecture Search

Designing a DNN is an iterative process that requires significant human expertise, research and computation time to tune the architecture and its hyperparameters to the target task and dataset, without any insurance that the resulting model will lead to acceptable results.

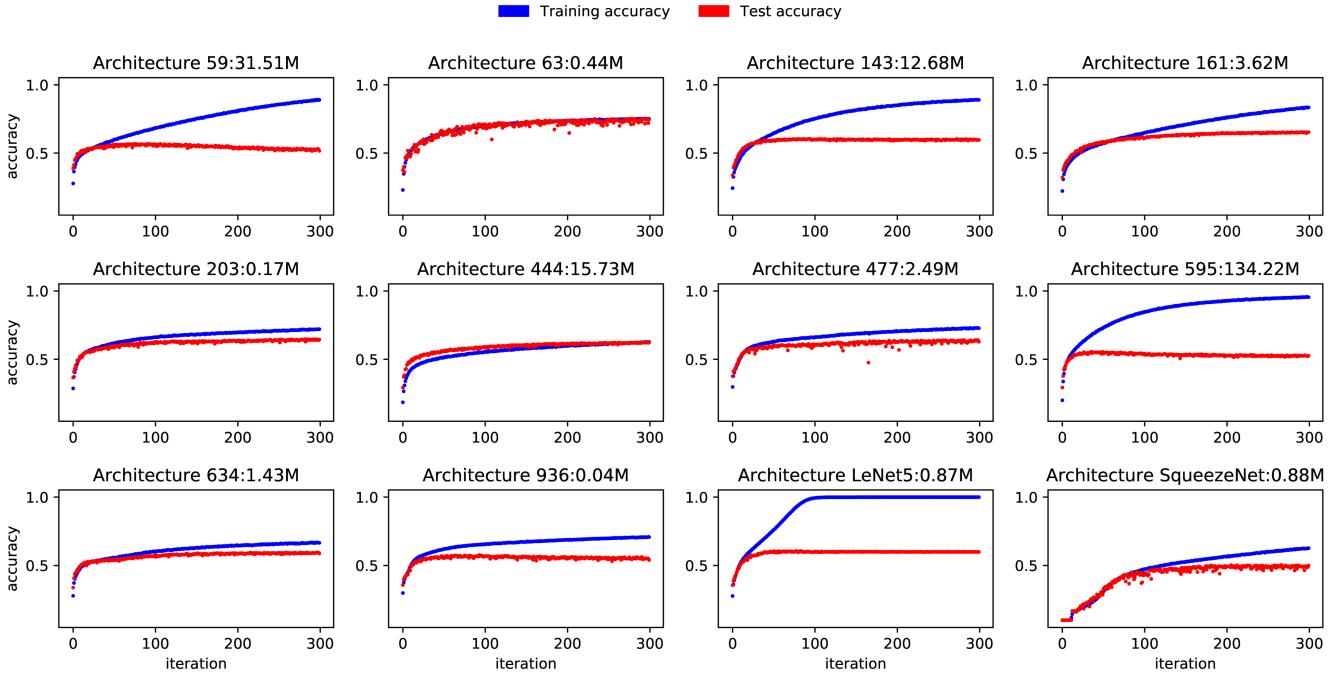
As of today, there is no consensus on a standard method to automatically generate these configurations. Automated Machine Learning (AutoML), the field that aims at providing engineers with automated means of building and deploying customized machine learning models, has, therefore, become a very important research topic. In particular, Neural Architecture Search (NAS) aims at identifying the best DNN architecture given specific tasks and performance metrics. Our work naturally falls into this area.

The time complexity of NAS approaches can be seen as  $O(N\bar{t})$  where  $N$  is the number of assessed architectures and  $\bar{t}$  the computation time required to assess every architecture. However, most techniques require a large  $N$  to reach good performances. The main approach consists of hyperparameter optimization using reinforcement learning, with techniques like MetaQNN [1], ENAS [26] and DeepArchitect[24].

MetaQNN uses Q-learning (a type of reinforcement learning) with Markov decision processes to iteratively select network layers and their parameters among a finite space. It defines every state as a layer with a set of hyperparameters and is thus limited to the traditional view of DNNs, which we generalize in our representation to account for the latest architectures. More precisely, it supports only convolution, polling, and dense layers, with a maximum of 2 dense layers and 12 overall. Another difference is the used search criterion: while theirs is based on the Q-learning metric, we rely on a sampling method driven by diversity. Although it achieves good performance, this method requires the evaluation of multiple thousands of architectures, leading to 8-10 days of calculation with 10 NVIDIA GPUs.

Efficient Neural Architecture Search (ENAS), based on PNAS [20], is a meta-learning approach that uses a recurrent neural network to train and reinforce a target CNN architecture. It starts from a simple component (also named a cell) and stacks it  $N$  times in order to build a final CNN. Their search problem is therefore reduced to identifying the best cell structure to replicate, rather than designing the entire convolutional network. This approach shares with our work the idea that we can achieve complex structures using a cellular approach. However, the components of their cells have no standard representation. An operator in an ENAS cell can be applied on one or multiple inputs and can lead to multiple outputs. By contrast, our technique relies on a finer-grained configuration process where the standard structure of all cells and blocks is customizable independently.

Finally, more flexible approaches to NAS are based on evolutionary algorithms (e.g. Genetic CNN [35]) but their performance and scalability are still to be assessed[37].



**Figure 8: Evolution of training and test accuracy over the training epochs. For each architecture, the title of the plot includes its size (in Million parameters) for reference**

## 7.2 Configuration sampling

Sampling is mainly used as an answer to the difficulty of assessing every software product[34], due to the explosion of the configuration space, when the number of features increases. While we considered diversity-based sampling [9], other techniques could be of interest and should be evaluated.

Most immediate solutions rely on uniform/random sampling, which unfortunately cannot scale to large feature models [27]. The extensive literature on product sampling includes many techniques to reduce the number of products to evaluate. Kim et al.[15] propose a technique based on static software analysis, where irrelevant features (i.e. with no impact) are removed. Johansen et al.[13] propose to split the SPL into sub-SPL to represent the knowledge of which interactions are prevalent.

Sampling can also be driven by feature coverage. T-wise algorithms [25] aims to cover all  $t$ -uples of features (e.g. pair-wise aims to cover every pair) at least one. Such techniques become more expensive as  $t$  increases, while they have to take into account the constraints of the feature model [3, 12].

## 8 CONCLUSION

We demonstrated how to model DNN architectures with an FM. Such modeling enables the application of variability management techniques on DNNs. Our variability model covers the most popular CNN architectures that have been engineered and hand-crafted in the past twenty years. Thanks to the compositional nature of FMs, our model can be extended easily to support more architectures. For instance, recurrent neural networks[29] have introduced a looping mechanism to allow reasoning over sequences of inputs, which is notably useful for speech recognition and language modeling.

Modeling such non-linear networks would require allowing cells' output to reach a preceding cell or block.

The hierarchical structure of FMs also allows the addition of new dimensions of variability. While the current FM focuses on the inner constituents of DNN architectures, we could extend it to include variability in the training setup (e.g. hyperparameters like learning rate, optimization method, data augmentation techniques and etc.) or in user preferences (performance metrics, explainability and etc.), and allow the possibility to restrict the configuration to a specific family of architectures (e.g., LeNet) through the dynamic addition of constraints.

An additional contribution of our paper regards the fully automated process that searches and deploys DNN architectures. Our search procedure relies on an out-of-the-box diversity-based configuration generation, which leads to architectures that outperform the state-of-the-art. Of course, our framework is not limited to diversity-based generation, which is used for demonstration purposes and can be extended with other configuration generation techniques. Nevertheless, since the training of hundreds of architectures demands tremendous computation resources, future research should focus on experimenting with additional search techniques, such as SATIBEA [8], or developing specialized techniques potentially combined with simulation and performance prediction methods [23] that have the potential to significantly reduce the required computations.

## 9 ACKNOWLEDGMENT:

Mike Papadakis is supported by the Luxembourg National Research Funds (FNR) C17/IS/11686509/CODEMATES.

## REFERENCES

- [1] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. 2017. Designing Neural Network Architectures using Reinforcement Learning. *CoRR* abs/1611.02167 (2017).
- [2] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.* 35, 6 (Sept. 2010), 615–636.
- [3] Mehra N. Borazjany, Linbin Yu, Yu Lei, Raghu Kacker, and D. Richard Kuhn. 2012. Combinatorial Testing of ACTS: A Case Study. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* (2012), 591–600.
- [4] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2013. Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-Features. In *ICSE'13*. IEEE, 472–481.
- [5] Krzysztof Czarnecki and Chang Hwan Peter Kim. 2005. Cardinality-based feature modeling and constraints: a progress report. In *International Workshop on Software Factories at OOPSLA'05*. ACM, ACM, San Diego, California, USA.
- [6] Robert Feldt and Simon M. Poulsen. 2017. Searching for test data with feature diversity. *CoRR* abs/1709.06017 (2017). arXiv:1709.06017 <http://arxiv.org/abs/1709.06017>
- [7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), 770–778.
- [8] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. 2015. Combining Multi-objective Search and Constraint Solving for Configuring Large Software Product Lines. In *Proceedings of ICSE '15*. IEEE Press, 517–528.
- [9] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Trans. Software Eng.* 40, 7 (2014), 650–670. <https://doi.org/10.1109/TSE.2014.2327020>
- [10] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. PLEDGE: A Product Line Editor and Test Generation Tool. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops (SPLC '13 Workshops)*. ACM, New York, NY, USA, 126–129. <https://doi.org/10.1145/2499777.2499778>
- [11] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2017. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2017).
- [12] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An Algorithm for Generating T-wise Covering Arrays from Large Feature Models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1 (SPLC '12)*. ACM, New York, NY, USA, 46–55. <https://doi.org/10.1145/2362536.2362547>
- [13] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. 2012. Generating Better Partial Covering Arrays by Modeling Weights on Sub-product Lines. In *Model Driven Engineering Languages and Systems*, Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 269–284.
- [14] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21.
- [15] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. 2011. Reducing Combinatorics in Testing Product Lines. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development (AOSD '11)*. ACM, New York, NY, USA, 57–68. <https://doi.org/10.1145/1960275.1960284>
- [16] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* (2014).
- [17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., USA, 1097–1105. <http://dl.acm.org/citation.cfm?id=2999134.2999257>
- [18] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86 (1998), 2278–2324.
- [19] Min Lin, Qiang Chen, and Shuicheng Yan. 2013. Network In Network. *CoRR* abs/1312.4400 (2013). arXiv:1312.4400 <http://arxiv.org/abs/1312.4400>
- [20] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Loddon Yuille, Jonathan Huang, and Kevin Murphy. 2018. Progressive Neural Architecture Search. In *ECCV*.
- [21] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 643–654. <https://doi.org/10.1145/2884781.2884793>
- [22] Raphael Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. 2011. A formal semantics for feature cardinalities in feature diagrams. In *VaMoS'11*. ACM, New York, NY, USA, 82–89.
- [23] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2018. Finding Faster Configurations using FLASH. *CoRR* abs/1801.02175 (2018). arXiv:1801.02175 <http://arxiv.org/abs/1801.02175>
- [24] Renato Negrinho and Geoff Gordon. 2017. DeepArchitect: Automatically Designing and Training Deep Architectures. (04 2017).
- [25] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. 2010. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST '10)*. IEEE Computer Society, Washington, DC, USA, 459–468. <https://doi.org/10.1109/ICST.2010.43>
- [26] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. 2018. Efficient Neural Architecture Search via Parameters Sharing. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, Stockholm, Sweden, 4095–4104. <http://proceedings.mlr.press/v80/pham18a.html>
- [27] Q. Plazar, M. Achér, G. Perrouin, X. Devroey, and M. Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *ICST '19 (to appear)*.
- [28] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *CoRR* abs/1609.04747 (2016). arXiv:1609.04747 <http://arxiv.org/abs/1609.04747>
- [29] Alex Sherstinsky. 2018. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network. *CoRR* abs/1808.03314 (2018).
- [30] Patrice Y. Simard, David Steinkraus, and John C. Platt. 2003. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. In *ICDAR*.
- [31] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), 1–9.
- [32] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (2014), 6:1–6:45.
- [33] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70 – 85.
- [34] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 1 (SPLC '18)*. ACM, New York, NY, USA, 1–13. <https://doi.org/10.1145/3233027.3233035>
- [35] Lingxi Xie and Alan Loddon Yuille. 2017. Genetic CNN. *2017 IEEE International Conference on Computer Vision (ICCV)* (2017), 1388–1397.
- [36] Ming Yuan and Yi Lin. 2006. Model Selection and Estimation in Regression With Grouped Variables. *Journal of the Royal Statistical Society Series B* 68 (02 2006), 49–67. <https://doi.org/10.1111/j.1467-9868.2005.00532.x>
- [37] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning Transferable Architectures for Scalable Image Recognition. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018), 8697–8710.

## ARTIFACT

We will introduce in this section the repository and the steps required to replicate the results presented in Section 6.

Our approach relies on a combination of a Java software and Python scripts and can be split in two parts:

- *PLEDGE Tool* for the generation of the products starting from a Feature Model representation. PLEDGE Tool can be downloaded at <https://github.com/christopherhenard/pledge>. It is an open source java software that can be used both with a graphical interface or using a command line interface.
- *FeatureNet Compiler* for the compilation of the products and conversion into working Tensorflow Neural networks. Once the models are parsed and compiled, they are also trained and their accuracy exported in a text report. The source code of this tool can be found at <https://github.com/christopherhenard/pledge> and contains a README explanation to run our software using a command line interface.

## Installation

Both software can be obtained by cloning their respective repository. PLEDGE requires the Java Runtime Environment (1.8+) to run while FeatureNet requires Python 3.5+, Tensorflow 1.12+ and Keras 2.24+. FeatureNet supports execution on both GPU and CPU, depending on your version of Tensorflow but we highly recommend running the experiments using GPU. All our experiments were run using a Tesla V100-SXM2-16GB GPU.

## Datasets

The experiments are based on the Feature Model representation that we handcrafted and is extensively presented in Section 3. In the folder "datasets" of the FeatureNet repository, you will find 2 features models, in the SPLOT XML format:

- **main\_full.xml** for the full feature model we proposed. It contains 6867 features.
- **main\_light.xml** A feature model that has been stripped of the heavy features in order to achieve reasonable calculation time. The restrictions that we applied are explained in section 5.2 and lead to 3296 features. This is the FM we used in our experiments.

Both features models can be opened using PLEDGE. We have also provided in the "datasets" folder some of the configuration files generated with PLEDGE when restricting sampling time to 60s. You will find 3 configurations files: **10Products.pdt**, **100Products.pdt** and **1000products.pdt**. The latter was used for the experimental study. Note that the Sampling process with PLEDGE uses a heuristic that leads to a different sampling for each run. This means that running PLEDGE on the same XML file will generate each time a different configuration file (files that have .pdt extension).

The experiments we run focused on image recognition datasets, mainly *Cifar10* and *MNIST*. These popular datasets are natively supported by the Keras Framework and selected a the compilation step.

## Compilation and training

Once you have obtained your .pdt file, you can use the python script **full.py** to compile the configuration into a working deep neural network:

```
$ python full.py -i <input_file> [-d <datasets> -t  
→ <training_epoch> ]
```

where *<input\_file>* is the path to the .pdt file without the file extension, *<datasets>* a comma separated string that defines the image datasets we are targeting. The possible values are *cifar*, *mnist* or *mnist,cifar*.

Finally *<training\_epoch>* defines how much training are we getting our models through. Having your models train over more epochs require more time to complete but will achieve better accuracy.

Note that *<input\_file>* is the only compulsory parameter, the other parameters are set by default in the **init** method of **full.py**.

## Results and plots

After every run of the **full.py** script, a new report file will be generated in the same folder as the input configuration file. The file will be in the form of

*<input\_file>\_<dataset>\_<training\_epoch>epochs\_1.txt* using the previous variables.

Every report file lists all the products converted into models, one line per product (= deep neural network). Each line is a space separated list containing:

- the index of the product in the original configuration file.
- the final test accuracy.
- a boolean of whether the training has been stopped because of a timeout (24h by default).
- the total training time in seconds.
- the number of trainable parameters of the model.
- the number of Flops (Float operations) required by the model.
- a history of the test accuracy by epoch. The values are separated by a # symbol.

These attributes are discussed in a separate section, for instance, the number of trainable parameters and Flops are an indicator of the complexity of the network. These reports can be used to generate all the plots and tables we provided in our paper. We also provide in the repository a utility script **plotter.py**. It can be used to extract statistics and plot some figures such as the training time, the accuracy distribution, the over-fitting. It only requires to replace the report path of with the path to your own files and run the desired plotting function.

## Help and Troubleshooting

From sharing our software with third parties, we notice that installation and the execution can fail for various reasons. The Wiki pages of our Github project introduce some examples and you can also go to <https://github.com/yamizi/FeatureNet/wiki/Troubleshooting> for a list of common issues. You can also notify us of issues and improvements with the Issue Tracker:  
<https://github.com/yamizi/FeatureNet/issues>.

# Piggyback IDE Support for Language Product Lines

Thomas Kühn

Software Technology Group

Technische Universität Dresden, Germany

[thomas.kuehn3@tu-dresden.de](mailto:thomas.kuehn3@tu-dresden.de)

Nicola Pirritano Giampietro

Computer Science Department

Università degli Studi di Milano, Italy

[nicola.pirritanogiampietro@studenti.unimi.it](mailto:nicola.pirritanogiampietro@studenti.unimi.it)

Walter Cazzola

Computer Science Department

Università degli Studi di Milano, Italy

[cazzola@di.unimi.it](mailto:cazzola@di.unimi.it)

Massimiliano Poggi

Computer Science Department

Università degli Studi di Milano, Italy

[massimiliano.poggi@studenti.unimi.it](mailto:massimiliano.poggi@studenti.unimi.it)

## ABSTRACT

The idea to treat domain-specific languages (DSL) as software product lines (SPL) of compilers/interpreters led to the introduction of language product lines (LPL). Although there exist various methodologies and tools for designing LPLs, they fail to provide *basic IDE services* for language variants—such as, syntax highlighting, auto completion, and debugging support—that programmers normally expect. While state-of-the-art language development tools permit the generation of basic IDE services for a specific language variant, most tools fail to consider and support reuse of basic IDE services of families of DSLs. Consequently, to provide basic IDE services for an LPL, one either generates them for the many language variants or designs a separate SPL of IDEs scattering language concerns. In contrast, we aim to piggyback basic IDE services on language features and provide an IDE for LPLs, which fosters their reuse when generating language variants. In detail, we extended the Neverlang language workbench to permit piggybacking syntax highlighting and debugging support on language components. Moreover, we developed an LPL-driven Eclipse-based plugin that includes a syntax highlighting editor and debugger for an LPL with piggybacked basic IDE services, i.e., where modular language features include the definition for syntax highlighting and debugging. Within this work, we introduce a general mechanism for fostering the basic IDE services’ reuse and demonstrate its feasibility by realizing context-aware syntax highlighting for a Java-based family of role-oriented programming languages and providing debugging support for the family of JavaScript-based languages.

## CCS CONCEPTS

- Software and its engineering → Domain specific languages; Integrated and visual development environments; Software product lines;

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC ’19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336301>

## KEYWORDS

Domain Specific Languages, Language Product Lines, Integrated Development Environment, Feature Modularity, Neverlang

### ACM Reference Format:

Thomas Kühn, Walter Cazzola, Nicola Pirritano Giampietro, and Massimiliano Poggi. 2019. Piggyback IDE Support for Language Product Lines. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC ’19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3336294.3336301>

## 1 INTRODUCTION

Tools for the development of *domain-specific languages* (DSL) and programming languages have made a major leap. Employing state-of-the-art language workbenches, designing custom DSLs or extensions to established languages became feasible for researchers and practitioners alike [15]. This led to a large amount of DSLs and multiple language extensions. Recently, researchers started investigating combining different language variants to develop families of DSLs as well as programming languages, e.g., [16, 25, 28]. Yet, as state-of-the-art language development tools have limited support for reusing language features<sup>1</sup> between different languages, they are not suitable to develop families of DSLs and programming languages. To overcome their limitations, researchers currently apply ideas from *software product lines* (SPL) to embrace the need for multiple variants of a language. Simply put, language families can be created as an SPL of compilers/interpreters, whereas each product corresponds to a language variant [23]. These product lines are denoted *language product lines* (LPL) and have been successfully employed for families of DSLs [17, 26, 33, 39, 40] and general purpose programming languages [6, 22, 23]. While these approaches introduced various methodologies and tools to design LPLs, they fail to provide an *integrated development environment* (IDE) for users of a selected language variant. This hinders the acceptance of these approaches among users and programmers, as they expect at least *basic IDE services*, such as syntax highlighting, auto completion, and debugging support. Although language development tools permit easy generation and/or implementation of basic IDE services for a specific DSL [15], it is infeasible to generate and/or implement them for all possible language variants. Similarly, designing and maintaining a separate SPL of IDEs corresponding to

---

<sup>1</sup>In line with [23], *language features* are either language constructs, e.g., *for loop*, or language concepts (without concrete syntax), e.g., *scope* and *coercion*.

the LPL leads to scattering of language concerns between two product lines. Arguably, a better solution follows “*the road to feature modularity*” [19] and packs a language feature together with its basic IDE services. Hence, we aim to piggyback basic IDE services on language features and, therefore, to automatically provide IDE support for LPLs, which fosters their reuse when generating language variants. In fact, we extended the Neverlang language workbench to piggyback syntax highlighting and debugging support on language components. Moreover, we propose an LPL-driven IDE that includes a syntax highlighting editor and debugger for arbitrary LPLs, whose basic IDE services ride piggyback on language features, i.e., where modular language components include the definition for syntax highlighting and debugging. To demonstrate its feasibility, we implemented an LPL-driven IDE, an Eclipse plugin providing a context-aware syntax highlighting editor and an LPL-driven debugger. To further illustrate its applicability for basic IDE services we employ two case studies. First, we showcase context-aware syntax highlighting for the family of Java-based role-oriented programming languages (RPLs) [22]. Second, we demonstrate adding and employing debugging support for the family of JavaScript-based languages [23]. In conclusion, piggybacking basic IDE services on language features enhances the modularity and reuse of basic IDE services and permits using an LPL-driven IDE.

The paper is structured as follows. In the beginning, Sect. 2 briefly introduces LPLs, the Neverlang language workbench, the corresponding AiDE LPL configurator, and the notion of IDE services. Sect. 3 proposes our approach for piggybacking basic IDE services on language components of LPLs. Sect. 4 presents our implementation. Sect. 5 outlines the case studies we conducted on two different LPLs. The paper is concluded by discussing related approaches in Sect. 6, and summarizing our results in Sect. 7. The Appendix highlights the artifact provided to reproduce our results.

## 2 PRELIMINARIES

### 2.1 Language Product Lines

The development of families of programming and domain-specific languages has gained popularity among researchers and practitioners, e.g., [16, 25, 28]. Following the ideas of software product lines (SPLs), a LPL facilitates the process of language development, which can be *customized* by selecting individual *features*. Similar to SPLs, a language could be designed to specifically suit a certain use case or application domain. For instance, authors [11, 31, 34] have shown that the many variants of state machine languages could be modeled as one single family of programming languages. Nonetheless, this is also true for *general-purpose programming languages*, from which *dialects* may be defined for DSL purposes. On one side, specialized versions of full-fledged programming languages can be employed in case of security purposes (e.g., Java Card [10]) or teaching programming [6, 13]. Language extension, on the other side, can be useful to embed new language features into an existing programming language, such as type-checked SQL queries [14].

### 2.2 Neverlang<sup>2</sup> in a Nutshell

The Neverlang [4, 8, 32] framework is built around the *language feature* concept. Language components, called *slices*, embodying the language features are developed as separate units that can be

```

1 module StateModule {
2   reference syntax {
3     State <- [StateName] "=: Expr";
4   }
5   role(evaluation) {
6     0. { newstate $1 state, $2 action); };
7   }
8 }
9
10 slice State {
11   concrete syntax from StateModule
12   module StateModule with role evaluation
13 }
14
15 language HooverLang {
16   slices Program StateDecl EventDecl TransDecl StateLst State
17   StateName EventList Event EventName Expr BExpr TransList
18   Transition Support
19   roles syntax < execution
20 }
21 }
```

**Listing 1: Syntax and semantics for the state concept.**

compiled and tested independently, enabling developers to share and reuse the same units across different language implementations. Here the development base unit is the **module** (Listing 1). A module may contain a **syntax** definition and/or semantic **roles**. A **role** defines actions that should be executed when some syntax is recognized, as prescribed by the *syntax-directed translation* technique [1]. Syntax definitions and semantic roles are tied together using **slices**. Let us consider the Neverlang realisation of the **State** module for the vacuum cleaner example shown in Listing 1.

Here the module **StateModule** declares a reference syntax for the state concept (Lines 2–4) and actions are attached to the nonterminals on the right of the production (Line 7). Semantic actions are attached to nonterminals by referring to their position in the grammar or through a label: numbering starts with 0 from the top left to the bottom right. Thus, the first **State** on Line 3 is referred to as 0, **StateName** as 1, and **Expr** as 2. The slice **State** declares in Line 12 that we will be using *this* syntax (which is the **concrete syntax**) in our language, with those particular semantics (Line 13). Finally, the **language** descriptor indicates (Lines 17–19) which slices are to be composed together to generate the language interpreter. Composition in Neverlang is, therefore, twofold: (1) between modules, which yields slices, and (2) between slices, which yields a language implementation. The composition result is independent of the order of specified slices. The grammars are merged to generate the complete language parser. Semantic actions are performed with respect to the parse tree of the input program; roles are executed in the order specified in the **roles** clause of the **language** descriptor. Please see [32] for further details.

### 2.3 AiDE<sup>3</sup> in a Nutshell

AiDE [22, 23] is an interactive configuration tool especially tailored to develop language product lines. It implements the method presented in [23, 33, 34] to automatically synthesize the *feature model* of a given language family out of language components developed with Neverlang [32]. Through its graphical user interface, depicted in Fig. 1, the user can explore the feature model, choose features,

<sup>2</sup> Available at <https://neverlang2.di.unimi.it>

<sup>3</sup> Available at <https://aide.di.unimi.it>

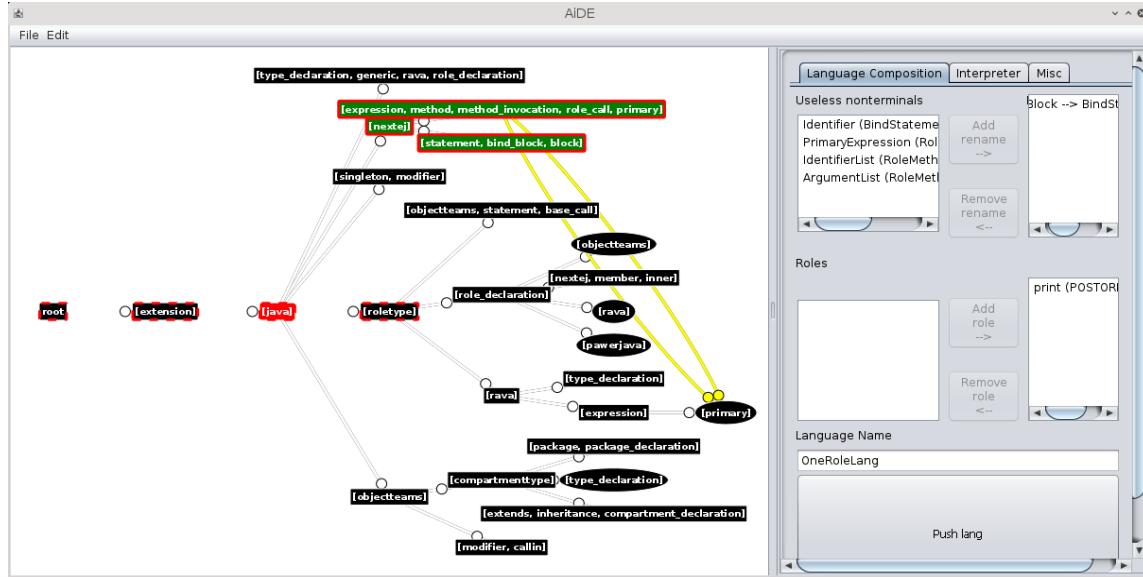


Figure 1: AiDE language configurator for the family of RPLs employed in [22].

create a language variant, and test it. Because feature models of LPLs tend to be large [22], AiDE initially shows the first level of the tree, however, allowing it to be expanded on demand. Moreover, while the user configures a language variant (or product), AiDE tracks all unresolved dependencies—i.e., all open nonterminals in the current selection—and provides the user with mechanisms, such as renaming, to bind them to other nonterminals already in the selection. Thus, users can easily resolve dependencies during the component selection. Another important feature of AiDE is its ability to dynamically update the language variant during its configuration. Whenever a valid configuration, i.e., one without unresolved dependencies, exists, the user can update the internal language variant and test it using the integrated command line interface of Neverlang. This, permits users to verify the consistency and test the behavior of the language variant under construction. Internally, AiDE updates the language descriptor maintained by the underlying Neverlang language development framework. When the language satisfies the expectations, a stable copy of the development environment is prepared and ready to be dispatched to any JVM compliant workstation. In sum, AiDE is able to guide users towards the generation of consistent language variants by supporting multiple dependency resolution strategies and continuous generation of the language’s compiler/interpreter.

## 2.4 Basic IDE Services

IDEs provide basic IDE services, like syntax highlighting, code completion, error marking, and debugging support. However, in line with Erdweg *et al.* [15], we distinguish between *syntactic services* and *semantic services*. Syntactic services only depend on the language’s syntax and are usually generated from its grammar, e.g., a syntax highlighting editor, an outline view, syntactic completion, and a pretty printer. In contrast, semantic services need to take the language’s semantics into account. Yet, only the simplest of them,

e.g., reference resolution and semantic completion, can be automatically generated from a specified language. Yet, more complex services require additional programming effort, such as refactoring, quick fixes, and debugging. Granted syntactic services are easier to integrate into an LPL, we argue that programmers equally require syntactic and semantic services from their IDE for a given language. Hence, we focus on syntax highlighting as a syntactic service and debugging as a complex semantic service.

## 3 PIGGYBACKING IDE SERVICES ON LANGUAGE COMPONENTS

Our main goal is to provide feature modularity and reuse for *basic IDE services* wrt. the language features they correspond to. To reach this, we have to reconsider feature modularity for languages. As Kästner *et al.* [19] argued, feature modularity relies on locality and cohesion, as well as on information hiding and encapsulation.

### 3.1 Integration into Language Product Lines

In case of LPLs, this entails that all information relevant for a language feature including the corresponding part of an IDE service should be part of one language component. This section shows how partial *IDE service specifications* relevant for a language feature can be added to each language component. Moreover, these specifications can be interpreted inside an LPL-driven IDE to dynamically provide the embedded syntactic and semantic services. Notably though, the possible integration of these service specifications into an LPL greatly depends on the nature of the IDE service.

### 3.2 Integration of Syntactic Services

Integrating a syntactic service into language components is straightforward. First, the service specification is decomposed wrt. to the syntactic production they contribute to.

```

1 module neverlang.statement.TryCatchStatement {
2   reference syntax {
3     Statement    ← TryStatement ;
4     TryStatement ← "try" Block CatchClause ;
5     TryStatement ← "try" Block CatchClause Finally ;
6     CatchClause  ← "catch" "(" CatchParameter ")" Block ;
7     Finally      ← "finally" Block ;
8     categories :
9       Keyword = { "try", "catch", "finally" }
10      with style "trycatch.json",
11      Brackets = { "(", ")" } with style "java.json" ;
12    }
13  /* ... */
14 }
```

Listing 2: Terminal categorization in a language component.

```
{ "Keyword": { "italic": true, "color": "#7f0000",
  "bold": true, "background": "#BF8F8F" } }
```

Listing 3: Style definition per category (trycatch.json).

Afterwards, the language component is extended to permit adding a partial specification of a corresponding syntactic service, as well as an interface to retrieve this specification. Finally, an LPL-driven IDE will query this interface of selected language components to provide the desired syntactic IDE service.

In case of syntax highlighting, the language component is extended with a style specification for each *terminal* in each production (right hand side). Although this approach would be feasible, if the style specification is compiled into the language components, the user cannot easily change the highlighting of a specific language feature. To remedy this, we permit language developers to define a custom category for each terminal in a language component. Simply put, a category is a name employed to retrieve a terminal's highlighting from a user controlled style file. Conversely, we augmented Neverlang *slices* to allow for defining categories for sets of terminals, as outlined in Listing 2 for the TryCatchStatement slice. In particular, the slice defines the category *Keywords* for the terminals *try*, *catch*, and *finally*, as well as the category *Brackets* for curly braces. Additionally, the *with style* clause denotes the user controlled file from where the highlighting style should be retrieved. Thus, while the IDE uses the default Java highlighting (*java.json*) for braces, it employs the style definition in *trycatch.json* for the keywords *try*, *catch*, and *finally*, shown in Listing 3. Specifically, the style definition does not only permit changing the font style, but also the font color and background color. Note that, *with style* enables overriding the default style of a category, such that the keywords of the try catch statement are highlighted in the context of this language component. In sum, while slices defines the categories for terminals, style files define the actual highlighting of terminals inside the textual editor.

### 3.3 Integration of Semantic Services

Compared to syntactic services, integrating semantic services into an LPL is complex, due to the fact that they depend on the language's semantics. Especially, this entails that the language's semantic actions must be intercepted and/or adapted by the IDE to

```

1 module neverlang.statement.Throwable {
2   reference syntax {
3     Statement    ← ThrowStatement;
4     ThrowStatement ← "throw" Expression SemiColonOpt;
5     /* ... */
6   }
7   role (debug) {
8     0 @{ $0.isExecutionStep = true; } .
9   }
10  role (evaluation) {
11    0 @{ /*...*/ } .
12    1 @{ /*...*/ } .
13  }
14 }
```

Listing 4: Adding debugging to a language component.

extract the required information for semantic services, such as reference resolution, error marking, and debugging support. Most generative LPL approaches, i.e., that generate interpreters/compilers of language variants, do not support the dynamic adaptation of their semantic actions. Yet, recently Cazzola and Shaqiri introduced *open programming language interpreters* for Neverlang-based LPLs, which enables language developers to adapt languages' semantic actions at predefined hooks [7]. As illustrated in Fig. 2, Neverlang encodes semantic actions, e.g., the evaluation of an expression, as attributes of nodes of the parse tree, such as  $E.val \leftarrow E_1.val * E_2.val$ , which are computed upon traversing the parse tree. By contrast, open interpreters allow language developers to attach language agents to hooks of syntax nodes (e.g.,  $E$ ) without requiring to change the language variant's implementation. Depending on the hook the attached agents are called before, instead, or after the node's evaluation [7].

Conversely, if the LPL is implemented as Neverlang's open programming language interpreter, it becomes possible to add a debugging service by adding a debugging language agent to an LPL. This can be done independently of the LPL, as long as the encompassed language components provide a debug role, which is evaluated before the actual evaluation role. Inside the debug role, the language developer only needs to add semantic actions to those productions that represent a computation step. These actions only mark a syntax node as an execution step for the debugger to distinguish between executable statements and non-executable language elements. For instance, the Throwable slice, depicted in Listing 4, for the *throw* statement. Here, the debug role only adds a semantic action to the ThrowStatement (Line 8) adding an attribute *isExecutionStep* set to true. During the evaluation the debugging agent checks whether this attribute is set for the syntax node, to decide when to suspend the debugger. The implementation of the language agents is outlined in Sect. 4.3.

## 4 A LANGUAGE PRODUCT LINE-DRIVEN IDE

After integrating basic IDE service specifications into an LPL, an LPL-driven IDE is needed to enable both researchers and practitioners to utilize them. This section describes a design process for a language engineer to configure and deploy a Neverlang-based LPL with included IDE services to an LPL-driven IDE. Moreover, we outline the implementation of our Eclipse plugin, denoted *textsfNeverlangIDE*, that facilitates our LPL-driven IDE featuring a context-aware syntax highlighting editor, as well as an LPL-driven debugger.

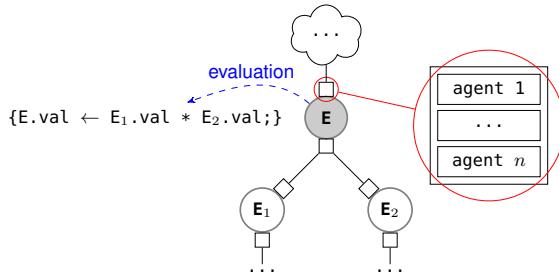


Figure 2: Parse tree with hooks and *before* agents, from [7].

#### 4.1 Language Configuration and Deployment

To benefit from the LPL-driven IDE, the language components must have been implemented as Neverlang slices. These slices must include suitable tags for the feature model generation [23], as well as style *categories* and *debug* roles for syntax highlighting and debugging support (cf. Sect. 3), respectively. As a result, such an LPL can be loaded into AiDE [23], which, in turn, generates the language’s feature model and guides the language engineer to choose and pick a valid language variant. After selecting a variant, AiDE generates a corresponding language descriptor. From this descriptor Neverlang compiles a corresponding compiler or interpreter for the language variant as an executable Java archive (jar). Additionally, this executable jar file also contains the style files, the *category* accessors, and the executable *debug* roles. Finally, it can be deployed to any JVM 1.8 compliant system where Neverlang is installed. Yet, to benefit from syntax highlighting and debugging support, an Eclipse ( $\geq 4.10.0$ ) instance is required where the NeverlangIDE plugin is installed. Now, only the location of the deployed jar must be announced to our plugin. After start up, the plugin loads all announced language variants, and allows users to benefit from the LPL-driven Neverlang Editor as well as the debugging support via Neverlang run configurations and an LPL-driven debugger. Henceforth, we will delve into their implementation.

#### 4.2 LPL-Driven Syntax Highlighting

To implement our LPL-driven text editor we extended Eclipse’s `TextEditor`. This editor requires an `ITokenScanner` to parse a given resource and decide how to highlight each lexical element. Fortunately, as any Neverlang language can be parsed using its extensible lexer [9, 32, cf. Lexter], we simply implemented the adapter to the `ITokenScanner`, sketched in Listing 5. In detail, the `LexterAdapter` queries for the language descriptor for the file extension of the underlying resource. Using this language descriptor both a Neverlang `lexer` and category-aware `parser` are retrieved. While the former is mandatory, the latter permits context-aware syntax highlighting. This becomes evident in the `nextToken` method, where the next token is retrieved from both the `lexer` (Line 31) and the `parser` (Line 33). In case the `category` retrieved for this token is defined, the `parser` will be requested to return a *contextual style* for this category (Line 38). Because each node in the parse tree is linked to its defining slice, it can expose its category and style definition. By contrast, if this style is `null` or the `parser` failed, syntax highlighting falls back to a style provided for the whole language, whereas the `tokenId` of the `lexer` is used as category (Line 42). Finally, the

```

1  public class LexterAdapter implements ITokenScanner {
2      private final LanguageProvider language;
3      private final LexterStream lexer;
4      private final StyledText editor;
5      private final SemanticHighlighterDexter parser;
6      private QualifiedToken lastToken, lastParserToken;
7      private int lastOffset;
8
9      public LexterAdapter(String lang, StyledText editor){
10         this.editor = editor;
11         this.language = LanguageProvider.getInstance(lang);
12         this.lexer = (LexterStream) lang.getDexter().getLexter();
13         this.parser =
14             new SemanticHighlighterDexter(language.buildLanguage());
15         this.lastToken = null;
16         this.lastOffset = -1;
17         /* Initialize lexer and parser...*/
18     }
19     @Override
20     public int getTokenLength(){
21         return (lastToken == null ? -1 : lastToken.text.length());
22     }
23     @Override
24     public int getTokenOffset(){
25         return (lastToken == null ? -1 : lastOffset);
26     }
27     @Override
28     public IToken nextToken(){
29         if (lastToken != null)
30             lastOffset += lastToken.text.length();
31         QualifiedToken nextToken = (QualifiedToken) lexer.getNext();
32         lastToken = nextToken;
33         lastParserToken = parser.nextToken();
34         String category = parser.lastTokenCategory();
35         TextAttribute attr = null;
36         /* Retrieve context-dependent style */
37         if (category != null)
38             attr = parser.contextualStyle(category);
39         /* Fallback to default language style */
40         if (attr == null)
41             attr = language.getTokenToAttributeMapper()
42                 .get(nextToken tokenId);
43         return new WrappedToken(nextToken, attr);
44     }
45     @Override
46     public void setRange(IDocument document, int offset, int length){
47         /*...*/
48     }
49 }
```

Listing 5: Implementation of the highlighter’s `TokenScanner`.

`nextToken` returns an Eclipse `IToken`, which wraps both the Neverlang token and corresponding style. Internally, these tokens are forwarded to Eclipse’s syntax highlighting `TextEditor`. For simplicity, we currently do not employ an intelligent damage, repair, and reconciliation strategy and always parse the whole document upon changes. Granted, this incurs a huge performance overhead, yet, suffices as a proof of concept. In sum, our LPL-driven editor does not only permit the syntax highlighting of the programs written in the selected language variant, but can also emphasize the provenance of language features, as will be demonstrated in Sect. 5.1.

#### 4.3 Language Agent-Based Debugging

To provide debugging support for an LPL-driven IDE, we first needed an LPL-driven debugger. Consequently, we implemented a prototypical, external debugger for Neverlang-based languages that can communicate with an external tool both synchronously and asynchronously.

```

1  public class DebugAgent extends Agent {
2      public enum State {
3          INIT, SUSPENDED, RUNNING, STEPPING, DISCONNECTED
4      }
5      private State state=State.INIT;
6      private BreakpointManager breakpoints=new BreakpointManager();
7      private DebugMessageSender debugSender=new DebugMessageSender();
8      private NodeInfo stepOverNode;
9
10     public DebugAgent(OpenNeverlang interpreter) {
11         super(interpreter); debugSender.send("started");
12     }
13     @Override
14     public void notifyEvent(ExecutionEvent executionEvent) {
15         switch (executionEvent) {
16             case FILE_LOADED:
17                 interpreter.registerAgent(this, new AnyPattern(), null,
18                     HookType.BEFORE_AND_AFTER); break;
19             case EXECUTION_FINISHED:
20                 this.terminate(); break;
21         }
22     }
23     /*...*/
24     private boolean suspendAt(NodeInfo node) {
25         return node.isExecutionStep()
26             && (state == State.SUSPENDED || state == State.STEPPING
27                 || breakpoints.hasAssociatedBreakpoint(node));
28     }
29     private void sendStopEvent(NodeInfo node) {/*...*/}
30     @Override
31     public void before(IPatternMatch iPatternMatch) {
32         NodeInfo node=interpreter.getCurrentNode();
33         handleAsyncCommands();
34         if (suspendAt(node)) {
35             sendStopEvent(node);
36             state=State.SUSPENDED;
37             handleSyncCommands(node);
38         }
39     }
40     @Override
41     public void after(IPatternMatch iPatternMatch) {
42         if (interpreter.getCurrentNode().equals(stepOverNode)) {
43             setState(DebugAgent.State.STEPPING);
44             stepOverNode = null;
45         }
46     }
47     @Override
48     public void interpreterChanged() {}
49 }
50 }
```

Listing 6: Excerpt of the debugging agent implementation.

This debugger supports the basic operations for setting up *breakpoints*, *suspending* an execution, *stepping into* or *stepping over* a suspended execution, as well as retrieving all variables (including values) of the current execution step. In its core the debugger adapts a provided Neverlang language variant by adding a DebugAgent, outlined in Listing 6. This agent is added to *each* node of the parse tree of the program to debug (Line 18) hooking itself before and after a node's evaluation (Line 19). Consequently, the agent has access to all values of a visited node in previous runs, such as the `isExecutionStep` property set in the `debug` role.

The DebugAgent's behavior is determined by its internal state ranging from the initial state INIT via the RUNNING state through to STEPPING and SUSPENDED to finally DISCONNECTED. The before hook method first handles all asynchronous requests (Line 33) and afterwards determines whether the debugger should be `suspendAt` the current node (Line 34). As defined in Lines 25–29, the debugger only stops at execution steps, i.e., nodes where `isExecutionStep`

has been set to `true`. Besides that, a running debugger is suspended if it reaches a user defined breakpoint, i.e., a node whose line number in the source code equals to the line number of a breakpoint. Otherwise, the debugger stops if it is either STEPPING or SUSPENDED. In case of STEPPING, the debugger will *step into* the next execution step found in the parse tree, e.g., the body of a for loop, the body of a called method or just the next statement. In contrast, *stepping over* requires to evaluate a complete subtree before suspending the debugger again. To achieve this, the debugger memorizes the `stepOverNode` for which a *step over* was issued, and resumes the debugger until this node is reached again in the `after` hook method (Lines 42–47). This ensures that the full subtree has been evaluated, before the debugger is set back into STEPPING state. Nonetheless, the language engineer can customize the debugger's behavior. In case of the `try` statement, the *step over* could either completely skip both the `try` and `catch` block by marking the `TryStatement` node as execution step or continue successively in the `try` and `catch Block` by marking each as execution step. As a result, the granularity and stepping behavior of the debugger is customizable by the language engineer through language components.

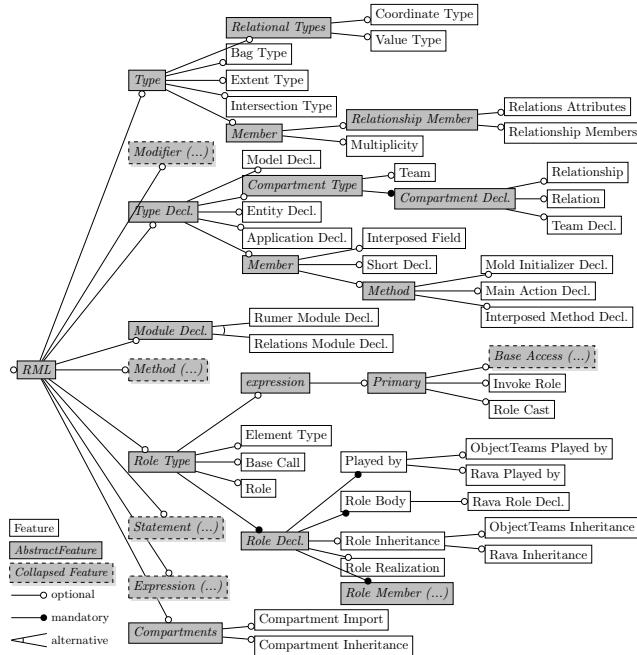
The LPL-driven debugger is a separate process that can communicate via TCP with any IDE providing a simple command interface and JSON-based data exchange. This simplified the integration of our debugger into Eclipse, as UI actions are delegated to the debugger, which returns serialized `VariableInfo` and `ValueInfo` objects for each variable, object, and object member in the scope of the current execution step to be shown in the *Variables* view. The `VariableInfo` interface describes a variable with its name, its type, and its value, i.e., a `ValueInfo` object. The `ValueInfo` interface, in turn, represents a value by means of its runtime type and its string representation. The LPL-driven debugger will triggered for Neverlang-based language variants, just like any Java program, by creating a run configuration for a program, selecting the corresponding language variant, and starting the debugger via the **Run ➔ Debug** menu entry. The LPL-driven editor permits to toggle breakpoints and shows the debugger's current position.

## 5 DEMONSTRATION CASE STUDIES

Admittedly, one could indicate the suitability of an LPL-driven IDE with toy examples, yet we believe that the benefits of piggybacking IDE services on LPLs shine when dealing with realistic LPLs. Consequently, we demonstrate the suitability of our LPL-driven syntax highlighting editor by augmenting the family of Java-based *role-oriented programming languages* (RPLs) [22] with style categories and style definitions for each language extension. Likewise, the suitability of our LPL-driven debugger is illustrated by adding debugging support to the family of JavaScript-based languages [23].

### 5.1 Family of Role-Oriented Programming Languages

The family of role-based modeling and programming languages was already identified in [25, 29]. A closer examination of RPLs in [22] revealed that most of them were extensions to Java. Accordingly, five of them were implemented as a Neverlang-based LPL [22], whereas each was implemented as an extension to a Neverlang-based Java parser, denoted Neverlang.Java.



**Figure 3: Generated feature model of role-based programming languages, from [22].**

**Feature Model and Language Decomposition.** The resulting family of RPLs follows a bottom-up LPL approach [22], which entails that the feature model, partially shown in Fig. 3, was generated by AiDE. The resulting feature model is rather big consisting of 73 features and 29 abstract features [22]. For brevity, several abstract features, e.g., *Modifier*, *Method*, and *Statement*, have been collapsed and all cross-tree constraints were omitted.

Thus, the resulting feature model only emphasizes language features of RPLs and not the underlying Java LPL.<sup>4</sup> Regardless, AiDE could still be employed to select valid language variants [22]. However, these language variants were hard to use, because different RPLs introduced different syntax with the same intentional semantics. Let us consider the various keywords to declare *roles*, e.g., **role**, **definerole**, **participants**, **class**. While a context-agnostic syntax highlighter could be defined, the user would still loose track of which language feature belongs to which language extension.

**Syntax Highlighting for Individual Language Features.** To approach this issue, syntax highlighting editors of state-of-the-art workbenches would not suffice. By contrast, our context-aware syntax highlighting editor for LPLs provides provenance for language features by highlighting related language features in the same style. In case of the family of RPLs, we added categories to all Neverlang slices in the LPL and included six style files within the LPL. One file defines the style for Neverlang Java and the others define individual styles for each of the five role-oriented language extensions, such that each language feature provided by an extension uses a distinguishing syntax highlighting. Specifically, Rava keywords have

<sup>4</sup>The full feature model for RPLs is available at [http://neverlang.di.unimi.it/aide/rplj\\_fm.pdf](http://neverlang.di.unimi.it/aide/rplj_fm.pdf).

**Table 1: Specifying syntax highlighting for Java and five language extensions, from [22].**

Language	Slices	Classes	LoC	Highlighting
Java	189	2	6843 (323)	208 (30)
Common	6	3	184 (850)	4
Rava	5	0	213	17 (12)
powerJava	7	0	243	14 (15)
OT/J	16	0	715	43 (15)
Rumer	31	5	1238 (146)	75 (16)
Relations	20	1	756 (33)	32 (16)

a lime background, powerJava violet, ObjectTeams/Java orange, Rumer cyan, and Relations blue. As shown in Table 1, piggybacking syntax highlighting on the family of RPLs required only 393 additional *lines of code* (LoC) and 74 lines in JSON style files (numbers in brackets). Naturally, we defined the nine categories of terminals according to the typical lexical tokenization of Java, i.e., Identifier, Type, Operator, Brackets, Keyword, String, Number, Boolean, and Character. However, most of the role-oriented extensions only override the styling of keywords, brackets, and operators, with the exception of Rumer that introduces the new type Extent. In sum, the manual implementation for adding syntax highlighting to the existing LPL for RPLs was limited and could technically be completely replaced by a code generator.

**Context-aware Syntax Highlighting Editor.** To test the LPL for RPLs, we generated multiple language variants ranging from the five RPL languages to a feature complete variant. The latter, allows for combining the various role declarations in one combined language. This language variant is used to showcase the contextual awareness of our syntax highlighting editor. After announcing this language variant to the NeverlangIDE plugin, we can open an example.rolejava file in Eclipse with the Neverlang Editor. In fact, right clicking on the file in Eclipse and selecting the Open With... ▶ Other context menu item, opens a dialog where our Neverlang Editor can be selected. Fig. 4 depicts a screenshot of the editor showing the content of the example.rolejava file. This file contains a role-based banking application implemented using role definitions (language features) provided by the different extensions. In contrast to typical syntax highlighting, for the first time, users are able to track the provenance of employed language features by means of a simple color coding. This helps to notice errors that would be otherwise missed. For instance, it becomes evident that a Rava role call (@INVOKEROLE) is used within a powerJava role (Lines 27–33). Thus, while users get insight into the provenance of employed language features, they are still able to customize their highlighting by modifying the style files.

## 5.2 Family of JavaScript-based Languages

The family of JavaScript-based languages, denoted NeverlangJS, was initially designed as a real world case study for Neverlang [32], yet it has proven its worth for gradually teaching JavaScript [6]. In detail, students were provided with specialized JavaScript variants, whereas each variant focuses on teaching another language feature, e.g., loops, recursion, exception handling, object orientation [6].

```

1 /* Pure Java */
2 class Account{
3     void increase(double amount){/*...*/}
4     void decrease(double amount){/*...*/}
5 }
6 class Person {/*...*/}
7
8 /* powerJava role interface */
9 @role Customer playedBy Person{/*...*/}
10 /* ObjectTeams context */
11 @team class Bank {
12     /* ObjectTeams role */
13     class Checking playedBy Account {
14         double limit = 500.0;
15         callin void decrease(Double amount){
16             if (amount<=limit) base.decrease(amount);
17         }
18     }
19     /* Rava role */
20     @ROLE Savings roleof Account {
21         double fee = 1.05;
22         void decrease(double a) {
23             @core Account().decrease(a * fee);
24         }
25     }
26     /* powerJava role */
27     @finerole BankCustomer realizes Customer {
28         boolean transfer(Account f, Account t, double a) {
29             /* Rava invocation */
30             if (f.plays(Savings))
31                 @INVOKEROLE(f, Account, Savings, decrease, a);
32             if (f.plays(Checking))
33                 @INVOKEROLE(f, Account, Checking, decrease, a);
34             t.increase(a);
35         }
36     }
37 }
38 /* Rumer relationship */
39 relationship Ownership participants(Person owner, Account owned) {
40     extent boolean addAccount(Person owner, Account owned) {
41         return these.add(new Ownership(owner,owned));
42     }
43 }

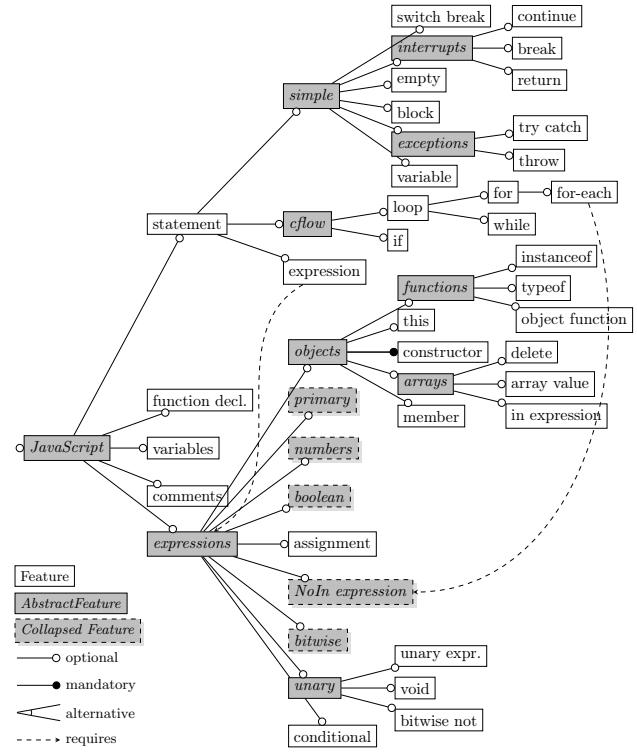
```

**Figure 4: Contextual highlighting of role-oriented extensions to Neverlang.JS.**

Although our experience showed the viability of this approach, we concede that our students struggled to find errors in their implementation, especially, in case of runtime errors. What our students missed most, was debugging support for the various language variants to easily set breakpoints and step through their running program. Conversely, this case study finally introduces debugging support to Neverlang.JS and all its variants. Moreover, the editing, the executing, and the debugging of JavaScript language variants is integrated into the Eclipse IDE.

**Feature Model and Language Decomposition.** Neverlang.JS is a fully decomposed version of JavaScript, whereas each module and slice corresponds to a specific language feature. Its implementation amounts to 3043 LoC and 228 production rules [32]. Each valid language variant is a functional JavaScript interpreter. In particular, the feature complete Neverlang.JS variant conforms to the ECMAScript 3 Language Specification (ECMA-262) and covers about 70% of the corresponding language specification [32]. Notably though, the remaining 30% amount to implementing built-in libraries, which is merely a technicality and is not required to showcase the debugging support.

Besides that, the Neverlang.JS LPL was one of the first LPLs to be configured by AiDE. Resulting from JavaScript’s complexity, AiDE generated a very large feature model with a maximum depth of 6 as well as 19 abstract features and 73 language features [23].



**Figure 5: Generated feature model of Neverlang.JS, from [23].**

Due to space restrictions, Fig. 5 shows a reduced feature model, where the abstract features *primary*, *numbers*, *boolean*, *NoIn expressions*, and *bitwise* have been collapsed.<sup>5</sup> In addition, the feature model captures that *constructors* are mandatory for *objects* and *for-each* loops depend on *NoIn expressions*, which define the *in* expression only inside *for-each* loops. However, while the Neverlang.JS LPL produces an interpreter for each member of the family of JavaScript-based languages [6], these interpreters lack direct debugging support.

**Adding Debugging to Neverlang.JS.** As outlined in Sect. 3.3 and Sect. 4.3, adding debugging support to an existing LPL includes two major steps: (1) marking execution steps and (2) exposing variables and their values in the current execution context. To improve usability, we kept debugging on the level of statements rather than expressions. In fact, most debuggers operate on this granularity, as stepping through expressions would be tedious.

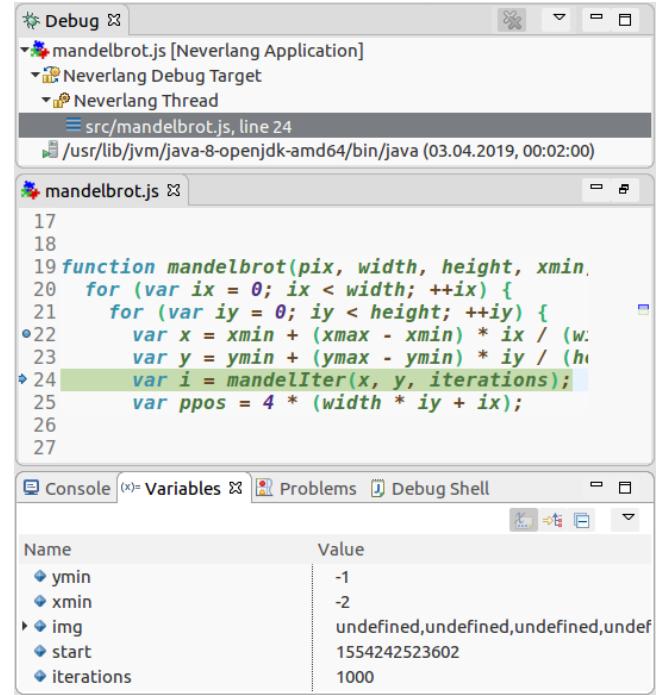
Consequently, in the first step, we extended all Neverlang slices that represent statements with a debug role with a corresponding action setting *isExecutionStep* to **true**. As depicted in Table 2, this includes statements, such as **if**, **switch**, **for**, **for-each**, as well as function calls, interrupts and conditional expressions, with the exception of the general *Block* and *Loop* statement. Besides marking execution steps for the debugging agent, the second step entails writing two classes implementing the *VariableInfo* and *ValueInfo* interface to expose variables and their values, respectively.

<sup>5</sup>A full version is available at [http://neverlang.di.unimi.it/aide/njs\\_graph.png](http://neverlang.di.unimi.it/aide/njs_graph.png).

**Table 2: Introducing debugging to the JavaScript-based language family, from [23].**

Features	Slices	LoC	Debugging
<b>Core</b>			
Language core	11	277	10
<b>Expressions</b>			
Arithmetic	3	128	-
Boolean	3	92	-
Relational	2	137	-
Conditional	1	32	7
Bitwise	5	216	-
Typing	2	65	-
Function call	2	113	12
Construct call	1	56	-
<b>Types</b>			<b>43</b>
String	1	21	-
Number	1	24	-
Boolean	1	23	-
RegExp	1	23	-
Object	4	189	30
Array	3	131	-
Function (definition)	2	100	-
This resolution	1	17	-
<b>Statements</b>			
Block Statement	1	32	-
If Statement	1	45	8
Switch Statement	1	102	5
(Loop Statements)	1	19	-
While Statement	1	50	8
For loop	1	57	7
For-each loop	1	113	13
(NoIn expressions integration)	11	305	16
Interrupts (break,continue,...)	3	74	15
Exception handling	2	122	5
<b>Variables</b>			
Variable assignment	5	226	11
Variable resolution	1	24	-
Symbol Table	0	230	8

In NeverlangJS these classes are implemented as `JSVariableInfo` and `JSValueInfo` and amount to 43 LoC (cf. *Types*). Note, these `VariableInfos` must be programmatically added to Neverlang's `VariableInfoManager` during the program's interpretation, e.g., whenever a variable is declared. Only then, the `DebugAgent` can access the variables visible in the current execution context, and finally expose them to Eclipse's `Variables` view. Consequently, this has been done for the language features *assignments*, *function calls*, and *objects* leading to an additional 53 LoC. Although, the first step can be automated by annotating production rules, the second step requires manual work to expose the interpreters internal data structures, such as, variables, objects, and arrays. In sum, the implementation overhead to add debugging support to the NeverlangJS LPL only amounts to 198 LoC, which is surprisingly small considering the benefit it provides to its users.

**Figure 6: Debugging a variant of Javascript in Eclipse.**

**Debugging JavaScript-based Language Variants.** By employing AiDE we have created and tested multiple specializations of JavaScript as well as a feature complete variant. Yet, the latter will be used henceforth to demonstrate our debugger. Just like a language variant with piggybacked syntax highlighting, a language variant with piggybacked debugging support must only be announced to the NeverlangIDE plugin. Then once Eclipse has started, users can create, open, and edit JavaScript files (\*.js). For simplicity, we created a `mandelbrot.js` file, which encompasses a simple JavaScript program that computes the Mandelbrot set, inspired by [5, Listing 7]. Now, we can execute this program by creating a new Neverlang *debug configuration* via the **Run → Debug Configuration...** menu item; selecting `neverlang.js.JSLang` from the set of announced language variants and the executable `mandelbrot.js`; and finally clicking on the **Debug** button. This will trigger the LPL-driven debugger by providing it with the language variant as well as the `mandelbrot.js`. Afterwards, Eclipse sets up a TCP connection to the debugger, which permits to directly interact with the debugger through Eclipse's debug interface. One such run is shown in Fig. 6, where a breakpoint was set in Line 22 via the context menu entry `ToggleBreakpoint`. This, in turn, suspended the debugger in Line 22, where a user clicked the `step over` button twice, such that the current location of the program is in Line 24. From this location, a user can either `step into` the implementation of the `mandelIter` or `step over` its execution such that the debugger reaches Line 25. In addition to the editor, the `Debug` view indicates both the current state of the debugger and the current location in the source code. Besides that, the `Variables` view (below) lists all variables in the scope of the current location and their corresponding values, whenever the debugger has been suspended or performed a step.

Finally, once the debugging is completed the debugger can be either terminated or resumed until it reaches another breakpoint. In conclusion, it is not just feasible to develop a Neverlang-based LPL with piggybacked debugging support, but the latter also provides similar usability as the standard Java debugger.

### 5.3 Discussion

In summary, the two presented demonstration studies provided evidence for the feasibility of piggybacking both syntactic and semantic IDE services on language components. Granted, one might argue that syntax highlighting is the simplest syntactic service, yet, our context-aware syntax highlighting editor exceeds editors generated by state-of-the-art language workbenches, especially as it can provide provenance of language features. Moreover, our demonstration study indicated that the implementation overhead for piggybacking syntax highlighting on Neverlang slices is negligible. Finally, as Neverlang's slices are typically tied to a reference syntax, other syntactic IDE services could be piggybacked in a similar way. That is, a minimal service specification could be added to slices, which is then retrieved and drives the behavior of a generic Eclipse-based View/Editor. Although no one will argue that *debugging* is a simple semantic IDE service, a similar argument can be drawn for other semantic IDE services. In fact, due to Neverlang's support for *open programming language interpreters* [7], other language agents could be attached to the interpreter, which can expose internal information required for a specific semantic IDE service. This information can then be used by an extended editor to provide the desired semantic IDE service. In conclusion, most of the effort will be put into providing an LPL-driven IDE service for a particular platform, e.g., Eclipse, whereas the implementation overhead for a language engineer will be limited to marking special syntactical elements or exposing semantic information via predefined interfaces.

## 6 RELATED WORK

Nowadays, the development of DSLs is a hot topic and a lot of research efforts are spent in this direction. Several language workbenches have been developed, such as Spoofax [38], MPS [35], MontiCore [21] and Melange [12]. All of these approaches provide a way to generate the IDE support for the DSL under development. In all these cases, the IDE support is tied to the developed DSL but its support is general and it does not exploit specific characteristics of the DSL. The IDE is automatically generated from templates neglecting the DSLs feature modularity, such that IDE services cannot be specified within DSL and reused from time to time as in this proposal. MontiCore [2, 3] and Melange [27] support the development of LPLs but their approaches only support basic syntax highlighting and code completion.

Besides that, EMFText [18] must also be highlighted, as another EMF-based tool [30] (like Xtext) that supports modular language implementation and explicitly supports the IDE generation for the developed languages. Even if EMFText does not explicitly consider variability in the IDE development, it has several commonalities with our LPL-driven IDE starting from the use of attributed

grammars for sharing information between languages and IDE implementation but also because of a specific DSL dedicated to the description of the IDE.

Looking at the current literature in software variability, some efforts have been made towards the description of families of graphical (modeling) editors, yet not for IDEs. Several tools of this kind appeared over the years, e.g., EuGENia [20], Graphiti [36] and Sirius [37]. EuGENia and Sirius are model-based, while Graphiti relies on Java code. Both EuGENia and Graphiti are based on code generation, whereas Sirius is dynamically interpreted. In general, all these approaches lack direct support for variability provided by SPL methodologies. Thus, variability must be hard coded leading to hard to maintain and hard to extend product lines.

In this respect, our latest contribution, FRaMED [24] properly supports variability in the development of graphical editors but not IDE. It focuses on the visualization/modeling of the program rather than providing support for its debugging or other typical needs of code-based development. FRaMED follows a top-down approach to the LPL construction whereas our LPLs follow a bottom-up approach; therefore the editor is generated through model composition instead of language component composition.

## 7 CONCLUSION

In this paper, we presented the idea of piggybacking portions of IDE services to the corresponding language component, enabling *de facto* the possibility of dealing with language variability *together* with IDE variability. Evidently, a language and its IDE are deeply interconnected and keeping their development separated violates feature modularity limiting the support an IDE could provide.

Conversely, our contributions are manifold. We introduced the concept of IDE services and how these can be piggybacked on language components. We explored the various kinds—syntactic and semantic services—of support an IDE provides and explained how these can be bound to the developed language variant through a LPL. Although the concept is generally applicable to all syntax-directed language workbenches, we have extended the Neverlang language workbench and applied it to families of real languages such as JavaScript and the Java-based role-oriented languages. The demonstration cases showed the feasibility of our approach, in general, as well as nice side-effects of context-aware syntax highlighting, such as tracing the provenance of language features.

Arguably, this work presented a proof-of-concept—even if it can work on real cases—so there is still space for improvement. In the future, we will integrate more syntactic and semantic IDE services, e.g., error marking, and semantic auto-completion. Moreover, we intend to permit dynamically reconfiguring the language variant within the IDE, such that both debugging and syntax-highlighting are dynamically adapted. Furthermore, we consider to support the *language server protocol* and widen the support for IDEs beyond Eclipse. Finally, we hope that other syntax-directed language development tools apply our idea.

## REFERENCES

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts.
- [2] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2018. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In *Proceedings of the 12th*

- International Workshop on Variability Modelling of Software Intensive Systems (VAMOS'18).* ACM, Madrid, Spain, 75–82.
- [3] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rümpe, and Andreas Wortmann. 2018. Modeling Language Variability with Reusable Language Components. In *Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC'18)*, Thorsten Berger and Paulo Borba (Eds.). ACM, Gothenburg, Sweden, 65–75.
- [4] Walter Cazzola. 2012. Domain-Specific Languages in Few Steps: The Neverlang Approach. In *Proceedings of the 11<sup>th</sup> International Conference on Software Composition (SC'12) (Lecture Notes in Computer Science 7306)*, Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book (Eds.). Springer, Prague, Czech Republic, 162–177.
- [5] Walter Cazzola, Ruzanna Chitchyan, Awais Rashid, and Albert Shaqiri. 2018.  $\mu$ -DSU: A Micro-Language Based Approach to Dynamic Software Updating. *Computer Languages, Systems & Structures* 51 (Jan. 2018), 71–89. <https://doi.org/10.1016/j.clsa.2017.07.003>
- [6] Walter Cazzola and Diego Mathias Olivares. 2016. Gradually Learning Programming Supported by a Growable Programming Language. *IEEE Transactions on Emerging Topics in Computing* 4, 3 (Sept. 2016), 404–415. <https://doi.org/10.1109/TETC.2015.2446192> Special Issue on Emerging Trends in Education.
- [7] Walter Cazzola and Albert Shaqiri. 2017. Open Programming Language Interpreters. *The Art, Science, and Engineering of Programming Journal* 1, 2 (April 2017), 5–15–34. <https://doi.org/10.22152/programming-journal.org/2017/1/5>
- [8] Walter Cazzola and Edoardo Vacchi. 2013. Neverlang 2: Componentised Language Development for the JVM. In *Proceedings of the 12<sup>th</sup> International Conference on Software Composition (SC'13) (Lecture Notes in Computer Science 8088)*, Walter Binder, Eric Bodden, and Welf Löwe (Eds.). Springer, Budapest, Hungary, 17–32.
- [9] Walter Cazzola and Edoardo Vacchi. 2014. On the Incremental Growth and Shrinkage of LR Goto-Graphs. *Acta Informatica* 51, 7 (Oct. 2014), 419–447. <https://doi.org/10.1007/s00236-014-0201-2>
- [10] Zhiqun Chen. 2000. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, Reading, MA, USA.
- [11] Michelle L. Crane and Juergen Dingel. 2005. UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05) (LNCS 3713)*, Lionel Briand and Clay Williams (Eds.). Springer, 97–112.
- [12] Thomas Dégueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: a Meta-Language for Modular and Reusable Development of DSLs. In *Proceedings of the 8th International Conference on Software Language Engineering (SLE'15)*, Davide Di Ruscio and Markus Völter (Eds.). ACM, Pittsburgh, PA, USA, 25–36.
- [13] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. 2012. Language Composition Untangled. In *Proceedings of the 12th Workshop on Language Description, Tools, and Applications (LDTA'12)*, Anthony Sloane and Suzana Andova (Eds.). ACM, Tallinn, Estonia.
- [14] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: Library-Based Syntactic Language extensibility. In *Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA'11)*. ACM, Portland, Oregon, USA, 391–406.
- [15] Sebastian Erdweg, Tjits van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Alex Kelly, Gabrieł Konat, Pedro J. Molina, Martin Palatinik, Risto Polohnen, Eugen Schindler, Clemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van del Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. *Computer Languages, Systems and Structures* 44 (Dec. 2015), 24–47.
- [16] Debasish Ghosh. 2011. DSL for the Uninitiated. *Commun. ACM* 54, 7 (July 2011), 44–50.
- [17] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gørán K. Olsen, and Andreas Svendsen. 2008. Adding Standardized Variability to Domain Specific Languages. In *Proceedings of the 12<sup>th</sup> International Software Product Line Conference (SPLC'08)*, Klaus Pohl and Birgit Geppert (Eds.). IEEE, Limerick, Ireland, 139–148.
- [18] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. 2011. Model-Based Language Engineering with EMFText. In *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'11) (LNCS 7680)*, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.). Springer, Braga, Portugal, 322–345.
- [19] Christian Kästner, Sven Apel, and Klaus Ostermann. 2011. The Road to Feature Modularity? In *Proceedings of the 15th International Software Product Line Conference (SPLC'11)*, Ina Schaefer, Isabel John, and Klaus Schmid (Eds.). ACM, Munich, Germany.
- [20] Dimitrios Kolovos, Antonio García-Domínguez, Louis M. Rose, and Richard F. Paige. 2017. EuGENia: Towards Disciplined and Automated Development of GMF-Based Graphical Model Editors. *Software System and Modeling* 16, 1 (Feb. 2017), 229–255.
- [21] Holger Krahn, Bernhard Rümpe, and Steven Vökel. 2010. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer* 12, 5 (Sept. 2010), 353–372.
- [22] Thomas Kühn and Walter Cazzola. 2016. Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines. In *Proceedings of the 20th International Software Product Line Conference (SPLC'16)*, Rick Rabiser and Bing Xie (Eds.). ACM, Beijing, China, 50–59.
- [23] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. 2015. Choosy and Picky: Configuration of Language Product Lines. In *Proceedings of the 19th International Software Product Line Conference (SPLC'15)*, Goetz Botterweck and Jules White (Eds.). ACM, Nashville, TN, USA, 71–80.
- [24] Thomas Kühn, Ivo Kassin, Walter Cazzola, and Uwe Aßmann. 2018. Modular Feature-Oriented Graphical Editor Product Lines. In *Proceedings of the 22nd International Software Product Line Conference (SPLC'18)*, Paulo Borba and Thorsten Berger (Eds.). ACM, Gothenburg, Sweden, 76–86.
- [25] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. 2014. A Metamodel Family for Role-Based Modeling and Programming Languages. In *Proceedings of the 7th International Conference Software Language Engineering (SLE'14) (Lecture Notes in Computer Science 8706)*, Benoit Combemale, David J. Pearce, Olivier Barais, and Jürgen Vinju (Eds.). Springer, Västerås, Sweden, 141–160.
- [26] Jörg Liebig, Rolf Daniel, and Sven Apel. 2013. Feature-Oriented Language Families: A Case Study. In *Proceedings of the 7<sup>th</sup> International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13)*, Philippe Collet and Klaus Schmid (Eds.). ACM, Pisa, Italy.
- [27] David Méndez-Acuña, José A. Galindo, Benoît Combemale, Arnaud Blouin, and Benoît Baudry. 2017. Reverse Engineering Language Product Lines from Existing DSL Variants. *Journal of Systems and Software* 133 (Nov. 2017), 145–158.
- [28] Karen Ng, Matt Warren, Peter Golde, and Anders Hejberg. 2011. *The Roslyn Project: Exposing the C# and VB Compiler's Code Analysis*. White Paper. Microsoft.
- [29] Friedrich Steimann. 2000. On the Representation of Roles in Object-Oriented and Conceptual Modelling. *Data and Knowledge Engineering* 35, 1 (Oct. 2000), 83–106.
- [30] Dave Steinberg, Dave Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework*. Addison-Wesley.
- [31] Laurence Tratt. 2008. Domain Specific Language Implementation Via Compile-Time Meta-Programming. *ACM Transactions on Programming Languages and Systems* 30, 6 (Oct. 2008), 31:1–31:40.
- [32] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A Framework for Feature-Oriented Language Development. *Computer Languages, Systems & Structures* 43, 3 (Oct. 2015), 1–40. <https://doi.org/10.1016/j.clsa.2015.02.001>
- [33] Edoardo Vacchi, Walter Cazzola, Benoît Combemale, and Mathieu Acher. 2014. Automating Variability Model Inference for Component-Based Language Implementations. In *Proceedings of the 18th International Software Product Line Conference (SPLC'14)*, Patrick Heymans and Julia Rubin (Eds.). ACM, Florence, Italy, 167–176.
- [34] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoît Combemale. 2013. Variability Support in Domain-Specific Language Development. In *Proceedings of the 6<sup>th</sup> International Conference on Software Language Engineering (SLE'13) (Lecture Notes in Computer Science 8225)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer, Indianapolis, USA, 76–95.
- [35] Markus Völter and Vaclav Pech. 2012. Language Modularity with the MPS Language Workbench. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE, Zürich, Switzerland, 1449–1450.
- [36] Vladimir Vujošić, Mirjana Maksimović, and Branko Perišić. 2014. Comparative Analysis of DSM Graphical Editor Frameworks: Graphiti vs. Sirius. In *Proceedings of the 23rd International Electrotechnical and Computer Science Conference (ERK'14)*. Portorož, Slovenia, 7–10.
- [37] Vladimir Vujošić, Mirjana Maksimović, and Branko Perišić. 2014. Sirius: A Rapid Development of DSM Graphical Editor. In *Proceedings of the IEEE 18th International Conference on Intelligent Engineering Systems (INES'14)*, Tamás Haidegger and Levente Kovács (Eds.). IEEE, Budapest, Hungary.
- [38] Guido H. Wachsmuth, Gabrieł D. P. Konat, and Eelco Visser. 2014. Language Design with the Spoofox Language Workbench. *IEEE Software* 31, 5 (Sept./Oct. 2014), 35–43.
- [39] Christian Wende, Nils Thieme, and Steffen Zschaler. 2009. A Role-Based Approach towards Modular Language Engineering. In *Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09) (Lecture Notes in Computer Science 5969)*, Mark van den Brand, Dragan Gasevic, and Jeff Gray (Eds.). Springer, Denver, CO, USA, 254–273.
- [40] Jules White, James H. Hill, Jeff Gray, Sumant Tambe, Aniruddha Gokhale, and Douglas C. Schmidt. 2009. Improving Domain-specific Language Reuse with Software Product-Line Configuration Techniques. *IEEE Software* 26, 4 (July-Aug. 2009), 47–53.

## A ARTIFACT

In addition to this publication, we provide an artifact to reproduce our results. In particular, we published a virtual machine packed with the ready to use syntax highlighting editor with debugging support, denoted NeverlangIDE. In detail, it is an Eclipse plugin that includes a syntax highlighting editor and debugger for two *language product lines* (LPL) with piggybacked basic IDE services, i.e., where modular language features include the definition for syntax highlighting and debugging.

### A.1 Installation and First Step

We provide our LPL-driven IDE as a *virtual machine* (VM), because it is supposed to work in 10 years from now. It was prepared for the virtualization environment VirtualBox and is installed, as follows:

- Download and install VirtualBox from their [website](#).<sup>6</sup>
- Download the NeverlangIDE [image \(~1.3 GB\)](#).<sup>7</sup>
- Open your VirtualBox and use **File ➔ Import Appliance** to select and import the downloaded file.
- Start the added NeverlangIDE virtual machine.

After the virtual machine is launched, double click on the NeverlangIDE icon on the desktop to start Eclipse with the NeverlangIDE plugin. You can use it to inspect the already opened JavaScript and RoleJava (\*.rolejava) files or debug the *Mandelbrot* script (*mandelbrot.js*). Because the configuration and generation of a language variant is too complicated to be explained in one page, we opted to provide you with two products of such LPLs showcasing syntax highlighting and debugging, respectively. Henceforth, we focus on these two usage scenarios.

### A.2 Debugging the JavaScript LPL

Once you have opened the editor *mandelbrot.js*, you can inspect the script computing the *Mandelbrot* set. It already contains *breakpoints* at Line 22 and 60, however, you can add and remove breakpoints by right-clicking on the line number and selecting *Toggle Breakpoint*. Currently, the Debug view and the Variables view are empty, they will be populated automatically during debugging.

- (1) Click on the Debug icon (alternatively, use **Run ➔ Debug**) to start debugging.
- (2) After a short while the Debug view is populated, with among others a “Neverlang Debug Target”. Click on the triangles to unfold the “Neverlang Thread” and finally “src/mandelbrot.js, line 60”.
- (3) Click on “src/mandelbrot.js, line 60” to update the editor and populate the Variables view. **Note, that otherwise Step-Into and Step-Over will not affect the editor and the view!**
- (4) Use the *Step-Into* icon until you jumped into the *mandelbrot* function (Line 20).
- (5) Use *Step-Over* or *Step-Into* as you like to proceed with the stepwise execution.
- (6) Continue the execution by clicking on the Resume icon, which halts the execution again on Line 22.

<sup>6</sup><https://www.virtualbox.org/>

<sup>7</sup><https://adapt-lab.di.unimi.it/NeverlangIDE.ova>

<sup>8</sup><https://neverlang2.di.unimi.it>

- (7) To complete the execution simple right-click on breakpoints (Line 22) and click on *Toggle Breakpoint*. Afterwards, click on *Resume* to complete the execution. (Alternatively, you can always click on *Terminate* to kill the current debug session.) A complete execution will emit the computed mandelbrot set as array and the required time on the *Console* view. A complete run inside the debugger will take a long time.

### A.3 Context-Aware Syntax Highlightin

To illustrate the context-aware syntax highlighting, five files of the family of Java-based role-oriented programming languages have been included; four of these are already open when Eclipse starts. The file *example.rolejava* showcases context-aware syntax highlighting in action—i.e., it shows some language features from different languages (selected through the LPL) living together with their original syntax highlighting. Whereas the remaining \*.rolejava files feature each different role-based programming language, e.g., Rava, PowerJava, ObjectTeams, from which the language features are selected. **Because the NeverlangIDE is running inside a VM and utilizes dynamic class loading, opening an editor takes a long time.**

As a usage scenario, we suggest you try copying role or team definitions from the various role-based programming languages into the combined *example.rolejava* file.

- (1) Open the *example.rolejava* editor and browse through the source code. Note that some keywords, e.g., *class*, are highlighted differently depending on their position.
- (2) Open the *rava.rolejava* editor (be patient). Select the role definition *Checking* (Lines 19–28) and copy it into Line 7 of the *example.rolejava* file. As a result, highlighting is retained, i.e., the keywords of Rava are highlighted in green.
- (3) Open the *powerjava.rolejava* editor (be patient). Select the role definition *CA* at Line 8 and copy it into Line 18 of the *example.rolejava* file. Note that this definition can only occur as innerclass, hence including it outside of the *Transaction* team, would result in a syntax error (in turn, disabling context-aware syntax highlighting).
- (4) Open the other editors by clicking on and selecting the corresponding file. Try copying other roles, teams or methods to the *example.rolejava* and explore the effects.

**Be aware that once the program contains a syntax error, the context-aware syntax highlighting falls back to the default highlighting of Java. In that case use Undo to revert the file to the original state.**

If you accidentally close one of the files, they can be opened through the **Project Explorer**. Just, open then “RoleJava” project and go to the “src” folder. It contains all \*.rolejava files, which can be opened by double clicking on them (be patient).

### A.4 More Information

Inside the virtual machine the folder *Artefact/Languages/* contains the two LPLs, there you can inspect the implementation, whereas the Neverlang files are gathered in the *nlg-src/* folder and auxiliary Java classes in the *src/* folder. More information on Neverlang, its development and its use can be found on its [website](#).<sup>8</sup>

# Industrial Perspective on Reuse of Safety Artifacts in Software Product Lines

Christian Wolschke

Martin Becker

Sören Schneickert

Rasmus Adler

FirstName.LastName@iese.fraunhofer.de

Fraunhofer Institute for Experimental Software  
Engineering IESE  
Kaiserslautern, Germany

John MacGregor

Robert Bosch GmbH

Renningen, Germany,

John.MacGregor@de.bosch.com

## ABSTRACT

In the future, safety-critical industrial products will have to be maintained and variants will have to be produced. In order to do this economically, the safety artifacts of the components should also be reused. At present, however, it is still unclear how this reuse could take place. Moreover this reuse is complicated, by the different situations in the various industries involved and by the corresponding standards applied.

Current industrial practice for certification processes relies on a component-based view of reuse. We investigate the possibilities of product lines with managed processes for reuse also across multiple domains.

In order to identify the challenges and possible solutions, we conducted interviews with industry partners from the domains of ICT, Rail, Automotive, and Industrial Automation, and from small- and medium-sized enterprises to large organizations. The semi-structured interviews identified the characteristics of current safety engineering processes, the handling of general variety and reuse, the approach followed for safety artifacts, and the need for improvement.

In addition, a detailed literature survey summarizes existing approaches. We investigate which modularity concepts exist for dealing with safety, how variability concepts integrate safety, by which means process models can consider safety, and how safety cases are evolved while maintenance takes place. An overview of similar research projects complements the analysis.

The identified challenges and potential solution proposals show how safety is related to Software Product Lines.

## CCS CONCEPTS

- Software and its engineering → Software safety; Abstraction, modeling and modularity;
- General and reference → Surveys and overviews; Empirical studies; Reference works.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336315>

## KEYWORDS

Safety Reuse, Product Line Certification, Open Source Certification, Modular Safety, Safety Standards

### ACM Reference Format:

Christian Wolschke, Martin Becker, Sören Schneickert, Rasmus Adler, and John MacGregor. 2019. Industrial Perspective on Reuse of Safety Artifacts in Software Product Lines. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3336294.3336315>

## 1 INTRODUCTION

Safety-critical software is becoming more complex due to the ever greater possibilities it offers for inter-connectivity as well as due to its increased computing power. Demand for building ecosystems that integrate software from multiple vendors while still fulfilling the safety goals of the domain is rising. The vendors can be classic component suppliers as known from the automotive domain, but may also include suppliers of operating systems like the Linux kernel which are based on open-source development and have no specific integrity focus.

There is a dichotomy between the standards being based on the development of one-off products and products in industrial production being inherently variable. Today, the required variation of safety arguments is achieved by adapting each system integrity argument to the overall ecosystem. Pre-engineered components can be designed well for specific safety requirements, if the domain and the expectations are mostly clear upfront. As these constraints are only fulfilled in very limited cases, we investigate which challenges regarding the certification of components and systems exist and which process- and product-related measures enable the reuse of components in safety-critical domains.

As traditional safety analysis lacks the notion of variability, systems prescribe integrity requirements for lower level components which must be designed to fulfill these requirements. With the advanced integration of external systems, the use of Commercial Off-The-Shelf (COTS) products and higher runtime adaptivity, systems require more flexible design methodologies.

The general problem has so far been mostly addressed only by academia. Land et al. [43] found that components need to provide safety-preserving interfaces with appropriate abstractions expressed by models. The integration has to trace the system-level hazards and contexts to the respective component design decisions

and assumptions. A component can be pre-certified or it has to be certifiable meaning it is presumed that the development process allows further certification activities.

The challenges posed by increased openness [28], Cyber-Physical-Systems [59], Industry 4.0 [38], the integration of software product lines (SPL) [42] continuous updates and the heterogeneous nature of systems [45] have been well explored in academia so far. However, the industry perspective is still missing.

In order to shed more light on the state of the practice for the reuse of safety artifacts in software product lines, we have investigated problems industry is facing nowadays and which future solution approaches are foreseen. The investigation was commissioned to explore how companies in different domains cope with the variability in their product lines. Commonalities are interesting insofar as they suggest potential for improving standards and certification procedures. Differences are interesting as they suggest the potential for improved support for certification. Based on semi-formal interviews performed with six industry partners across multiple application domains and based on a literature and project survey, the paper provides the following *contributions*:

- it provides an overview on related work, including a literature survey on the state of the art,
- it presents along the interviews best practices that are already in place and the challenges considered most important by the interview partners.
- it discusses how safety artifacts could be integrated to product line settings.

The paper first provides a detailed analysis of the applicable standards and state of the art respectively, in sections 2 and 3. As its core contribution, the main findings of the interviews are presented in section 4. The following discussion (section 5) identifies how research could contribute to industrial needs. Finally, conclusions are drawn and an outlook to future work is given in section 6).

## 2 APPLICABLE STANDARDS

Safety certification requires adequately addressing the risks in products. In order to define what "adequate" means, we list safety-relevant standards in this section and analyze how they deal with the reuse of safety artifacts [46, 57]. The included domains are: *General E/E/PE systems, Automotive, Railway, Medical, Avionics and Machinery*:

**E/E/PE systems** . The functional safety requirements for general E/E/PE systems (electrical/ electronic/ programmable electronic systems) are specified in IEC 61508 [8]. The standard gives general recommendations, which are refined in other standards for their respective domains. If functionality is reused within a safety function, the integrator needs to assure that the reused parts do not violate the recommendations. The standard assumes that hardware and software developers consider the safety requirements in their architecture, as well as in the selection of methods, techniques, and tools.

This means that component developers should be aware of the safety requirements and that the reuse of generic components is not foreseen. A specific experience on how to

achieve reuse across multiple business units of ABB is given in [36].

**Automotive** . The functional safety of road vehicles is detailed in ISO 26262 [3]. The standard refines the general IEC 61508 standard for the specific needs of the domain of road vehicles. It was originally only intended for cars with a maximum weight of 3.5 tons, but was extended in 2018 to cover trucks and motorbikes as well. In terms of reuse, the ISO 26262 standard also requires that the safety requirements are elicited via a hazard and risk analysis and then distributed to the individual components. The component suppliers are responsible for fulfilling these requirements and for providing the necessary documentation. The focus on one domain enables the development of components that are built on assumed safety requirements and integrity levels, so-called Safety Elements out of Context (SEooC). This approach allows suppliers to develop components independent of a particular automaker. During system integration it has to be verified that the assumed safety requirements meet the actual safety requirements. In order to achieve a particular automotive safety integrity level (ASIL, which is a refinement of SIL), the standard allows the so-called ASIL decomposition. Instead of implementing one costly safety function with higher ASIL requirements, the function is split up into several safety functions with lower ASIL, which are cheaper to achieve. The standard also provides a Proven-in-Use argumentation, which allows integrating components that are already used in the field for which appropriate observations exist.

**Railway** . In the railway domain, the standards EN 50126 [4], EN 50128 [5] and EN 50129 [6] define how safety should be ensured for trains and signaling facilities. The standards demand evidence of quality and safety management to be provided, as well as functional and technical safety. The safety cases defined in the standards are:

- Generic product safety cases, which can be reused for different independent applications.
- Generic application safety cases, which can be reused for application types with common functions.
- Specific application safety cases, which are used for only one particular installation.

**Medical** . In the medical domain, the standard IEC 60601-1 [7] defines the safety-relevant issues for electronic medical devices. The standard requires components to comply with their safety requirements. Dedicated reuse issues are not considered in the standard.

**Avionics** . In order to receive regulatory approval, the software components of commercial airplanes have to comply with RTCA-DO-178C (also named EUROCAE-ED-12C) [27]. The idea is to use certification-relevant artifacts of components in a system's certification. The integrators use the artifacts, identify gaps regarding the safety requirements, and perform the remaining activities such as integration testing.

**Machinery** The refinement of IEC 61508 for machines is the standard IEC 62061 [9]. The standard assumes a modular decomposition of the system into subsystems. Along with the decomposition the safety requirements from the system level are also refined into safety requirements of individual

subsystems. The reuse of existing subsystems is not part of the standard.

The investigation of safety standards reveals the current understanding that safety requirements are first formulated for the overall system and then broken down into subsystems and finally into individual components. The safety requirements of components can also be used to out-source development processes to suppliers. In the case of pre-assumed safety requirements (such as SEooC), suppliers may already provide COTS components, which meet the safety requirements of the OEM. The current development envisages a high level of involvement of the suppliers in the integration of the components into the overall system.

### 3 STATE OF THE ART

As mentioned in the previous section, the state of the practice in safety certification is still (mainly) based on the assumption that a complete product is developed from scratch and certified by refining the safety requirements into individual components. The safety requirements concern the measurable quality of the product as well as the development process. In recent years, research has focused on the following questions:

- How to formulate the concept of modularity in the context of safety so that reuse of safety-critical components is possible (see section 3.1)?
- How to integrate safety aspects in variant-rich systems (see section 3.2)?
- How to integrate safety requirements formally in development process models (see section 3.3)?
- How to deal with safety in case of system evolution (see section 3.4)?

#### 3.1 Modularity and Safety

The increase in complexity is addressed by the notion of modularity. First we consider how components can be interconnected via the Demand and Guarantee Approach, then we will discuss the generation of appropriate component fault models by applying Fault Tree Analysis (FTA), and finally we will introduce the concept of safety supervisors.

*Demand and Guarantee Approach.* Rushby and Miner [54] extend the notion of Design by Contract to the safety domain. Components should specify which demands/assumptions they have and which guarantees they can provide if the demands are fulfilled. This demand & guarantee approach enables safety engineers to build up a hierarchy of safety cases. A formal description of this hierarchy is given in [24].

Originally, it was intended to perform integration at design time to allow engineers to also consider informal specifications. In order to reduce the time needed for integration, to decrease the number of interface errors (leading to an increase in quality), and to enable automatic checks at configuration time or even at runtime, the notion of Assume and Guarantee relations has been extended. The problem that assumptions created by one organization might not fit the guarantees created by another development organization is stated in [33]. The OPENCOSS [1] project addresses such issues by promoting a certification approach that incorporates the reuse of

safety arguments, safety evidence and contextual information about system components. The lack of formality in contract descriptions is addressed by [17] as part of the project. Their solution is to describe the demand and guarantee relations formally so that automatic reasoning can be established. The specification language employs temporal logic and satisfiability modulo theories (SMT) for the verification.

*Component Fault Trees (CFTs).* The concept of modularity with respect to Fault Tree Analysis was introduced as Component Fault Trees (CFT) by [41] and extended to Component Integrated Component Fault Trees ( $C^2FT$ ) by [26]. The idea is to develop fault models for components and link them to the architecture of the system. This eases the reuse of components along with their linked CFT as well as the evolution of the overall system, as the system and the Fault Tree are modularized in a similar manner.

*Safety Supervisor.* In the case of untrusted systems, such as autonomous AI-based systems, safety can be ensured by means of a safety supervisor, which offers a fail-safe behavior if the underlying AI-based system violates a safety goal [10].

#### 3.2 Variability and Safety

A core principle of product lines is to support adequate variability in core assets in order to adapt them efficiently to application-specific needs. "A Classification and Survey of Analysis Strategies for Software Product Lines" is presented by Thüm et al. [58]. They show that traditional product line engineering is aware of the need to perform quality analyses and that product lines are often also used in safety-critical domains. Nevertheless, safety cases are not considered in their work. The inclusion of safety in model-based development of product lines is investigated by a systematic mapping study of Baumgart [14]. The study found that about 40% of the work is concerned with the integration of hazard analysis, in particular with the techniques of (component) fault trees.

Variability may impact safety in the following ways:

- One domain - different needs - different applications** if system variants are created to satisfy different needs within one safety domain (section 3.2.1), or
- One domain - same needs - different applications** if a new different system variant should still address the same safety goals (section 3.2.2), or
- Different domain - different needs - same applications** if an already developed system variant is supposed to address different safety domains (section 3.2.3), or
- Tools** if tools and processes in the developing organization have to be adapted (section 3.2.4).

On the one hand, product line engineering needs to be able to certify that the assets fulfill the same safety goals for different variants, on the other hand, variants may require addressing different safety goals.

**3.2.1 Variability in Hazard and Risk Analysis.** Products based on safety-critical software product lines may have different contexts in which they are used. Hence, different hazard and risk analyses (HARA) have to be performed. Oliveira et al. [22] provide a systematic approach dealing with these impacts of variability. Their idea is to integrate the context explicitly into the feature model,

so that the selection of the supported context has an influence on the safety considerations (like HARA and FTA) as well as on the system design.

**3.2.2 Variants addressing the same safety goals.** If all variants need to fulfill the same safety goal, it is possible to build up a safety argumentation by using the Goal Structuring Notation (GSN [34]). In the approach of Habil [35], undeveloped goals are used as connection points in product line engineering. In the application engineering activity, these connection points are extended by the safety arguments of the selected components. Hutchesson and McDermid [37] extend this idea by collecting lifecycle evidence.

In the pSafeCer project [2] (funding frameworks: EU Artemis & Swedish SYNOPSIS) the combination of Model-based Development (MBD) and Component-based Software Engineering (CBSE) was examined. Conmy and Bate [18] investigate how to use GSN in a defined way by applying patterns.

The DECOS project uses the notion of service for separation of concerns [12]. The approach separates Generic Safety Case for the DECOS core services, Generic Safety Case for DECOS high-level services and Safety Cases for DECOS applications. This enables a direct link to a service oriented architecture.

Another approach is to model the difference of variants explicitly and to perform the analysis for a family of systems. Seidl et al. [56] present an approach for extending Component Fault Diagrams (CFDs). The product line is modeled via deltas, which state which elements (i.e., Events or Gates) and which ports are added, removed or changed. New variants take the existing CFD and extend their new part.

Braga et al. [16] propose modeling the relationship between a required certification level and features of a product line explicitly. Based on the required certification, those features can be selected that determine a set of selectable products. The link to the development process is necessary to keep the safety case compliant with the current software version. In the case of model-based development, it is possible to apply the Atlas Transformation Language (ATL) to define how the transformation from development model to safety case model can be performed automatically [44]. In order to systematically include safety considerations at the domain level of a product line, Olivera et al. [49] propose to use integrated HARA and fault analysis.

**3.2.3 Variants addressing different safety goals.** While the goal in the previous section was to show that a new, enhanced, or better configuration of a product still conforms to the same safety goals, this section analyzes the problems that arise if an existing product is to be re-used in different contexts with different safety requirements and regulations. Practical problems with the reuse of an automotive hall-effect sensor are reported by van Landuyt in [60]. The sensors used in connection with windshield wipers have less and different safety hazards than the sensors in brake pedals. Their proposal is to foresee different ASIL requirements already at the domain level in design and verification by providing reusable tests. At the application level, the corresponding variant is selected and the necessary safety assessments and verifications are performed. As part of the OPENCOSS project a systematic ISO 26262 compliant reuse approach for SEooC was proposed by Ruiz et al. [53]. The authors explain how ISO 26262 can be implemented

as a reference frame in the OPENCOSS Framework. Using the example of an Electronic Parking System, they explain how a generic safety case is produced with assumptions and public claims, and how the compliance match is performed during integration. According to Dehlinger and Lutz combining FTA with product-line engineering enables the reuse of safety artifacts [23]. In order to get modularity  $C^2FTs$  (see section 3.1) has been employed [25]. Their extension to variability is possible via so-called 150% models. This means that the  $C^2FTs$  model contains all possible variants and that the application engineers only need to subtract those parts that are not relevant for their system variant. As part of the SPES XT project Kaessmeyer et al. [40] presented an approach for integrating components in an automotive context. The approach formalizes the change impact analysis by using model-based approaches. An application of the approach at AUDI AG showed promising results. A detailed analysis of how model-based approaches, safety, and variability can be combined is given in Baumgart [14]. Procter et al. [50] propose defining Error Types, which enable the analysis of fault propagation with a family of platform-based systems.

**3.2.4 Combining tools and processes for safety critical variants.** The usage of pure::variants in the context of ISO 26262 safety artifacts is demonstrated by a study made with the Schulze et al. [55]. The integration of pure::variants to the SAFE framework is described by Oertel et al. [48]. Their approach consists of modeling the presence/absence of assumptions and guarantees based on the selection of features. BigLever's GEARS tool supports the application of software product line integration in safety-critical domains like automotive and defense [62].

### 3.3 Process Models and Safety

The process for creating systems has a tremendous impact on the quality of a product including safety. Hence, established processes should be reused and also adapted to meet the given safety goals. ABB established a reusable safety lifecycle model to meet SIL-compliant requirements [36]. This idea has been extended to the notion of process lines. Bringing together certification according to ISO 26262 and the process of Automotive SPICE is explained in [47]. The idea is to use generic safety cases of the reused artifacts and to tailor the process- and product-based argument in application development. The integration of formal process models in SPEM 2.0 and the integration of GSN is explained in [29]. An application to the Railway domain is presented in [30]. The reuse of tools involved in certification across domains is addressed in [32]. The explicit modeling of commonalities and variabilities between the standards to enable reuse and flexible process derivation is addressed in [31]. This approach was implemented by Javed and Gallina [39]. In the implementation they allow to combine SPEM models with models written in Common Variability Language (CVL). The Eclipse Process Framework (EPF) is used for the implementation.

The aspect that globally distributed development teams can work together on safety cases is discussed in [11]. They propose a cloud-based software that supports the generation of product-based and process-based safety arguments.

### 3.4 Evolution of Safety Cases

The evolution of a system requires adapting the safety cases accordingly. Besides publications on workshops about the mining of traceability information [15], and the model-based integration of traceability for safety cases [61], the OPENCOSS project is the most prominent project for providing means for safety assurance evolution. OPENCOSS is an EU project aiming at[21]:

- creation of a common certification conceptual framework,
- compositional certification,
- evolutionary chain of evidence,
- transparent certification process, and
- compliance-aware development process

La Vara et. al. [19] identify the frequency of change situations, and the tools used. The results show that in most projects re-certification of an existing system takes place after some modification, and that in new systems existing components are reused rather often. Impact analysis usually provides semi-automated recommendations only for source code. In the case of other specifications there is either decision support or only fully manual analysis is available. Hence the authors worked on formalizing the safety notation to increase the possibilities of automation. The Structured Assurance Case Metamodel (SACM) [20] provides a standard for the notation and evaluation of demand and guarantee contracts, which has been implemented as a model-based Eclipse plugin solution in the OpenCert platform [21]. The systematic reuse of safety certification artifacts is enabled by the usage of dedicated meta-models and mapping relations [51]. The authors identify the principles for the reuse of safety certification artifacts across standards and domains, such as "Safety certification artifact intent", "Equivalence mapping between standards", and "Compliance mapping". An experience report on cross-domain assurance involving the reuse of software component development is given in [52]. The usage of natural language in the OPENCOSS context was examined by Attwood and Kelly [13].

## 4 INDUSTRY INTERVIEWS

Based on previous project experiences, the challenges with the reuse of components in safety-critical industry contexts are well-known to us. Some of these are:

- How can safety cases be reused?
- How can the re-certification effort be kept as low as possible?
- How can a generic system element or system family be certified efficiently?

Companies across different industries and domains of all organizational sizes are facing serious challenges in this regard. In order to explore industry experience with problems, ideas, and solutions regarding the reuse of safety-relevant artifacts, we conducted interviews with safety experts employed in domains Automotive, Advanced Driver Assistance Systems (ADAS), Rail, Commercial Vehicles, Industrial Automation, Energy, and ICT.

The interview survey is intended to give answers on how different industries developing safety-critical Cyber-Physical Systems are affected by these challenges. The survey should also reveal possible approaches and best practices in development to this end. This survey provides an overview of the current state of development regarding the target topic.

### 4.1 Interview Partners

Fraunhofer IESE selected contacts known to be facing re-use challenges and having safety-relevant products. The selection limits the validity of the study, as the few companies or departments involved are European and only represent a few domains. The study was intended to assess the feasibility and direction of future work in this area. Table 1 characterizes the differences of companies in the interviews.

No.	Application Domain	#Product variants per year	#Employees	Level
1	ICT	$\leq 100$	$\approx 50$	Solution and Tool provider
2	Rail	$\approx 2 - 3$	10K-100K	Solution and Service provider
3	Automotive	$\approx 1$	Not specified	OEM
4	Rail, Trucks	$\approx 100$	10K-100K	Tier1 supplier
5	Automotive, ADAS	$\leq 100$	100K-200K	Integrator (Tier1)
6	Industrial Autom.	1K-10K	10K-100K	Tier1&2

Table 1: Characteristics of companies interviewed

### 4.2 Interview Preparation

The layout of the interview survey provided a framework for semi-structured interviews of 30-60 minutes each.

The interviews were structured as follows:

- 1) *Organizational description (5-10 minutes).* The organization is analyzed according to its domain, size, and products.
- 2) *Safety engineering characterization (5-10 minutes).* The current safety certification processes are characterized. The relevant standards and their impacts on the organization's business are elaborated.
- 3) *General variety and reuse characterization (10-15 minutes).* It is determined how many variants the company is producing, and which degree and type of reuse is already in place. The length of the evolution cycle, the reuse methodologies applied as well as the integration of open-source software are explored.
- 4) *Approach pursued for safety artifacts (5-10 minutes).* The impact of reuse on safety considerations is elicited. The degree of variability of the safety artifacts and the impact on the certification processes are surveyed.
- 5) *Need for improvement (5-10 minutes).* Finally the interviewees are asked to give their opinion on how their organization, application domain, overall industry, standardization organization and research may evolve to improve the handling of safety-relevant artifacts.

In the following, the results of a selection of the individual interviews conducted will be presented. The need for improvement will be shown in the summary of all interviews.

### 4.3 Interview 1 (ICT)

**Organizational Description** The company is a tool provider in the field of safety analysis, functional safety, and reliability of Cyber-Physical-Systems. The company unit develops and offers a model-based tool for safety and reliability analysis that supports the product certification of its customers. It comes in variants from standard products to projected customizations. While the company develops in an agile manner, the tool supports all variants of life-cycle models (e.g. (iterative) waterfall, (iterative) agile).

**Safety engineering characterization** A particular focus is on the automotive domain and ISO 26262 (but not limited to these). The tool has been subsequently extended to support other domains like Avionic, Railway and Machinery; the tool itself is qualified according to ISO 26262; the engineering process for the tool is assessed via SPICE; no homologation is required.

**General variety and reuse characterization** The standard product accounts for about 90% of sales; about 8% are related to configured tool variants that differ by approx. 30%, which is mainly caused by added or dropped functionality; new technologies drive a once-per-year release cycle. The company uses modular building kits and configurable components for reuse, and deploys generic interfaces and connectors. Eclipse is also deployed. Open-source software is validated via functional tests (no code checks).

**Approach pursued for safety artifacts** Re-qualification of the tool is done using a tool qualification kit with only little effort needed (modularization - 150% approach). From the viewpoint of the company, automated MBSE, model based comparison and formalized safety are proven concepts for keeping the certification effort for their customers low. Customers develop worst-case scenarios and special variant cases to cover all usage scenarios for generic elements and reuse most notably FME(D)A at the hardware level; GSN is only used sporadically, CFTs are not in use yet. From the tool vendor perspective the concepts w.r.t. reuse proposed by the standards are not concrete enough and therefore less supportive. The criticality of changes/variations concerning safety depends on the safety concept.

### 4.4 Interview 2 (Rail)

**Organizational Description** The company produces train systems & software, electrical and mechanical components, and contributes to rail automation. The organization's contributions to safety include methods, tools and architectures; i.e., software plays a large role in the products. The products are offered as standard, configurable, and projected variants. The company needs to qualify/certify its own products and systems as well as supplier products. The organization deploys various life cycle models for engineering processes.

**Safety engineering characterization** The company must adhere to all relevant safety standards in the Rail domain and its engineering processes are mandatorily assessed by external inspectors. Accredited inspectors also assess the developed software. Many product parts are certified based on European regulations, while others are subject to national regulations.

**General variety and reuse characterization** The company provides 2-3 product variants per year (with up to 30% variance), and releases new products every two or three years. Reuse comes in the form of clone & own and platform technology, and applies to the component level (over the full range of component variability) as well as the system level. The company does not deploy compatible, interchangeable components. It uses a lot of open-source software (e.g. operating systems) for which the relevant EN 50128 sections give hints on how to safeguard the safety level. Which concrete methods are used to this end depends on the application purpose.

**Approach pursued for safety artifacts** All kind of safety artifacts are affected by the product variability. To keep the effort with certification low, the company formulates worst-case scenarios, uses Component Fault Trees (CFTs), and in a few cases defined variation points in safety artifacts. However there is no specific method with respect to FMEAs. The expert-based worst-case scenarios are also the foundation for generic elements to be certified. The company reuses safety artifacts in numerous ways (the more concrete the artifacts are, the better). To assure that reused artifacts fit to the new context, the company consults the customer and deploys Safety-related Application Conditions (SACs). The organization deploys corrective, adaptive, perfective, and preventive maintenance, but a re-certification of the product is needed only for adaptive and perfective maintenance.

### 4.5 Interview 3 (Automotive)

**Organizational Description** The company is an OEM in the Automotive domain and produces vehicles for which almost all components are purchased. For the organization software has increasing importance, mainly driven by the increasing number and importance of assistance functions. The vehicles can be configured, but configurations currently do not apply to safety-critical components. The company aims to pursue a top-down waterfall process but in practice, e.g. components are bought and system level requirements are adapted.

**Safety engineering characterization** Safety engineering mostly concerns software-based functionality. Therefore, ISO 26262 dictates assessments of the product and the engineering process. The product is self-certified, and external consultants are involved for this purpose. For some active safety systems (e.g. restraint systems), obligations regarding homologation exist.

**General variety and reuse characterization** The company does new product releases every two or three years, and currently provides one vehicle variant per year (w.r.t. safety-related variation). Re-use that takes place at component level

(e.g. ECUs, sensors) is performed in a clone & own style. As almost all components are purchased, many come from different suppliers and must be interchangeable. The company does not use open-source software, as all purchased components come with software included.

**Approach pursued for safety artifacts** The company reuses component fault trees for the adaptation to new usage contexts. Variability currently has no safety-related impact on the company's product. The company deploys the SEooC concept, but the huge differences between the tiers and the way they apply this concept constitutes an obstacle. The level of detail as well as the tooling or formats used differ greatly.

#### 4.6 Interview 4 (Rail, Trucks)

**Organizational Description** The company produces subsystems in the Railway and Truck domain. Complete systems need to be certified (SW & HW) while complying with the Technical Specifications for Interoperability demanded by the EU. Software plays an increasing role in the company's business, but currently accounts for less than 20% of the engineering budget, and still only comes bundled with hardware. The company configures its braking systems to the customers' needs (organized in projects), and deploys an iterative waterfall model with some agile methods in use.

**Safety engineering characterization** The company must adhere to all relevant safety standards in the Rail domain as well as to machinery and manufacturing directives and deploys CMMI upon customer request. The engineering process is mandatorily assessed by external assessors. Mechanical, electronic and software products are assessed separately and go through an obligatory external certification process. In the future safety certificates for railway undertakings shall be valid throughout the EU.

**General variety and reuse characterization** For some subsystems the company produces roughly 100 variants per year (30-50% variance) built by (re)using configurable and standard products on all levels (component to system), and different software bundles, plus adaptations and extensions. New systems are released approximately every 3 years; for electronics and software, a 0.5-1 year cycle applies. The company buys electronic/hardware components and tests them and mainly writes the software itself. The company reuses open-source software to a very limited extent (mainly OS components), because the software needs to be qualified before deployment (acc. EN 50126), which often leads to the same amount of effort as if the software had been developed internally. Therefore, mostly only prototypes for demonstration purposes use open-source software.

**Approach pursued for safety artifacts** All of the company's safety artifacts are affected by product variability. In addition adaptations are needed based on different assessment procedures worldwide. The aim is to have generic descriptions with nation-specific adaptation. Model-based approaches are seen as a possible solution, but may take a lot of time. The company tries to keep the certification effort low by

using onion models and generic safety concepts for the hardware, and deploying configurable software applications. In the company, safety cases and fault trees (no CFTs so far) are re-used, with generic approaches strongly supported by the standards. Reuse of FMEA depends on fault trees and safety cases. Use case specifics and worst-case scenarios are assessed for adaptation to new usage contexts. The impact of corrective maintenance is considered beforehand.

#### 4.7 Interview 5 (Automotive, ADAS)

**Organizational Description** The company works on components in the functional chain of sensing, controlling and actuating (e.g. smart cameras, ECUs, steering systems). The company does not certify at the vehicle level, but has to certify its own products. Software plays the biggest role in the products of the company, accounting for about 70-80%. Depending on the newness of a certain function, the company develops its products in projects and as configurable solutions. There are more or less no standard products due to the newness of the ADAS technology in the Automotive domain. The company currently mainly follows an iterative V-cycle model but is transitioning to an iterative agile model.

**Safety engineering characterization** The company mainly adheres to ISO/PAS 21448 (Sotif) and ISO 26262. The engineering process is assessed with Automotive SPICE (and similar standards). The products are certified through homologation by a trusted partner; no other certifications are carried out. In some cases the company also organizes certification in accredited laboratories.

**General variety and reuse characterization** The company produces up to 20 variants per year with a variance of 15-80% in the reused components. The evolution cycle takes about 3 years, and is mainly driven by the customers. The company normally follows the clone & own approach; in hardware development a platform-based approach is used. Managed re-use strategies are in preparation and being rolled out. Currently no open-source software is deployed, though the feasibility of the integration of open-source operating systems into a safety-critical environment is being investigated.

**Approach pursued for safety artifacts** All safety artifacts used are affected by the product variability. All the artifacts are reused (carried over from a similar case) and adapted after an impact analysis and evaluation of the deltas between the base product and its variant. The re-certification effort is kept low by putting the focus on essential aspects, and by using an engineering-knowledge-based impact analysis (e.g., w.r.t. interdependencies). A model-based approach is on the company's road-map. The company certifies individual products, yet with new functionality the SEooC concept is deployed by exploiting domain and expert knowledge as well as information on existing use cases. According to the company the standards support reuse quite well (e.g. change impact analysis), yet this also depends on the customer. Changes are made when bug-fixes are needed or when, driven by the OEMs, model-upgrade adaptations have to be made. This has the same impact as if the product itself would have changed.

## 4.8 Interview 6 (Industrial Automation)

**Organizational Description** The company is operating on Tier 1&2 and produces motor controls for drives, motor drives (also motor-independent), frequency converters, and compressors. Software already plays a large and still increasing role with a current overall proportion of approximately 75%. The company provides standard products, and also sells its products via projects; but it mainly distributes configurable solutions to be configured by the company and the customer. The company tries to integrate agile methods into its engineering processes basing on iterative agile or waterfall models.

**Safety engineering characterization** The company unit adheres to UL 1998 and is supposed to adhere to ISO/TS 16949. There is mostly no safety certification for the company's products, however. Regarding the assessment of the engineering processes, the focus is mainly on production, not on the software. The company integrates external assessors to this end. Products are certified based on UL standards for the electrical safety of the hardware. There are no obligations regarding the involvement of external organizations for assessments.

**General variety and reuse characterization** Several 1000 variants per year have to be handled by the company, with most variance being driven by different functionality or different power size. The main driver for the evolution are changes in the market, with continuous feature requests for minor changes and the necessity to support new motor types about every 2 years. At the same time, the underlying technology remains rather stable. The major product is featurized, and is developed in a product line setup, whereas a minor branch product (customized drives) is developed in a clone & own style. The granularity of reuse varies across projects. There is reuse at the system level as well as modular design, which allows for smaller reuse assets. The company does deploy open-source software, but not on a regular basis; no safety approval is demanded for the deployed open-source software.

**Approach pursued for safety artifacts** There is reuse of safety artifacts across products for "comparable" system elements. In principle the artifacts are reused taking a clone & own approach relying on a sound domain knowledge. Where deployed software is safety-related, the qualification is reduced to only those parts of the software that make up the safety-critical part. Re-qualification is needed only if there is a proven change in the software. Modular design helps the company with reuse and reduces efforts; best practices include keeping the number of dependencies at a minimum and making them one-directional where possible. Another recommendation is to deploy the KISS principle. The company strives to design and deploy generic sub-components for certification but this is still work in progress. The standards in the field support the reuse of hardware components sufficiently, but regarding software components the standards are rather vague. Most maintenance work conducted is corrective or adaptive (40% bug fixes, 50% new features).

In only very few cases does a perfective maintenance take place, and there is no preventive maintenance. For changed safety-relevant components re-certification is demanded.

## 4.9 Summary of Interviews

The interviewed companies' application domains range from Automotive, ADAS, Rail, Commercial Vehicles, Industrial Automation, Energy, and ICT to Systems Engineering in a broader sense. They all have to adhere to domain-specific standards, which are refinements of the basic functional safety standard ISO 61508. The companies need to qualify and certify their own products and systems as well as supplier products, and contribute to the market with methods, tools and architectures that are deployed in safety-critical contexts. Software plays an increasingly important role here, as even companies that started with 100% mechanical engineering are on the brink of adding 100% software products to their portfolio. The organizations deploy various life-cycle models for engineering processes with a tendency towards iterative procedures mixed with agile methods that are adapted to the organizations' specific needs.

Whether (accredited) inspectors from trusted partners have to be involved in qualification, certification, or assessments strongly depends on the regulations with which the companies have to comply in the respective application domains. The regulations also differ in the national contexts. Where allowed, the companies self-certify their products but involve external experts for safeguarding purposes. The same holds for assessments of engineering processes with the deployment of (A)SPICE or CMMI.

Variants from standard products to configurable or parameterizable products to projected customizations exist in the safety context - with different implications. For the reuse of safety artifacts most companies rely on a clone & own approach, as it is easy to handle (at least in the beginning) and offers some advantages in project driven solutions. Platform approaches are used only rarely, as the certification of generic components poses some challenges for the companies. Open-source software is only rarely in use because the qualification efforts would be too high to gain a benefit over commercial software. Only open-source operating systems seem to be used more often in a safety-critical context, while other types of software are mainly deployed in a non-safety context or for prototyping. In supporting contexts (e.g., tools for developing safety artifacts), open-source software can be validated through functional tests, which is less effort prone.

All companies deploy some kind of reuse for their safety artifacts. Mostly the artifacts are reused by carrying them over from a similar case, and adapting them after an impact analysis. Most often worst-case scenarios and special variant cases are developed to cover all usage scenarios for generic elements. FME(D)As are reused most notably at the hardware level, GSN is only in little use, fault trees are used, but CFTs are rarely in use yet. To handle generic and worst-case scenarios, customers or domain experts are consulted; also, the SEooC concept and SACs are in use. Most companies do not plan for maintenance, as it is seen as change request with all the re-certification effort in the loop. The companies differ in their opinion as to whether the existing standards provide sufficient help with reuse endeavors.

In the interviews the companies also named possible improvements for their organizations and application domains regarding the reuse of safety and security artifacts. Among the most prominent are the following:

**Broaden the usage of model-based approaches.** Deployment

of e.g. model-based safety cases, ready-made safety cases (templates), and patterns directly aim at supporting the reuse of safety artifacts.

**Broaden the usage of CFTs.** CFTs broaden the possibilities of reuse by making it possible to modularize the fault analysis according to the underlying architecture.

**Include variation points** in the safety cases that are dependent on variation points in the product-determining artifacts. This clearly traces changes in a product to its safety artifacts and keeps dependencies at a minimum. Regarding safety artifacts containing variation points, the interviewees gave some recommendations on how to improve the handling. (1) Often, communication between project partners with respect to variation is difficult, as there is no "common language". Variation models are not widely used, and comparability is low. To improve this, research on and implementation of commonly accepted variation models would be very helpful. (2) Develop pre-defined solutions for specific variation problems to simplify the specification, as existing comprehensive models for variation are too complex to work with. (3) Develop a more specific, more concrete description of variance between a basic product and its variant. (4) Reduce dependency on variation points in the product, and therefore, in its aligned safety artifacts.

**During development**, focus on modularization, dependencies, cohesion, and design rules. Development should be aligned with the architectural disciplines, and dependencies should be reduced and simplified.

**The transition** from platform-based development to software product lines is seen as a possible improvement. This will change the certification process, as it will be necessary to be able to certify core assets, but afterwards there will be a simplified certification when these are deployed in the products.

**Develop** generic safety concepts (150%)

**Enhance** the concept of Safety Element out of Context (SEooC) to be more standardized.

**Modularize** the safety concepts, develop/improve reuse concepts, and develop stronger formalization of the concepts.

**Deploy** a clearly defined change management that encompasses the following steps: identify changes, plan changes, and assess changes.

**When purchasing components**, consider functional safety systematically beforehand.

## 5 IDENTIFIED CHALLENGES AND SOLUTION PROPOSAL

The integration of safety artifacts to reuse repositories implies several challenges and improvement potentials. Although the number of interviews is limited and further research is appreciated, we gained insight to current problems.

As future research we identified the following challenges as indicated by the points in Figure 1. The research will concentrate to which degree the safety engineering should be part of the product line or application development (Point 1). The design within the product line development (Point 2) affects how the application development will reuse the artifacts (Point 3). The application development will perform the missing safety analysis (Point 4) and play the results back to the product line development (Point 5). On the long term the concept of product line certification will also have an impact on standard committees and the applicable safety standards (Point 6).

### 5.1 Safety Responsibility: Application vs. Product Line (Point 1)

Safety engineering activities are based on the corresponding standards. They can be performed in product line as well as in application development. It depends on the safety guarantees a product line gives.

### 5.2 Product Line Certification vs. Reusable Component Certification (Point 2)

The generation of a generic safety case models for reusable components could serve as means to build safety cases for the integrating application more efficiently. The definition of patterns, taxonomies and ontologies for fault models would support this approach. The components may be certified by proving adherence to certain safety integrity levels (SIL) and suited process documentation (i.e. used tools, degree of testing, etc.)

In case of product lines dedicated to safety-critical domains, the safety case can be produced for the entire product line while also ensuring that all instances fulfill the safety case.

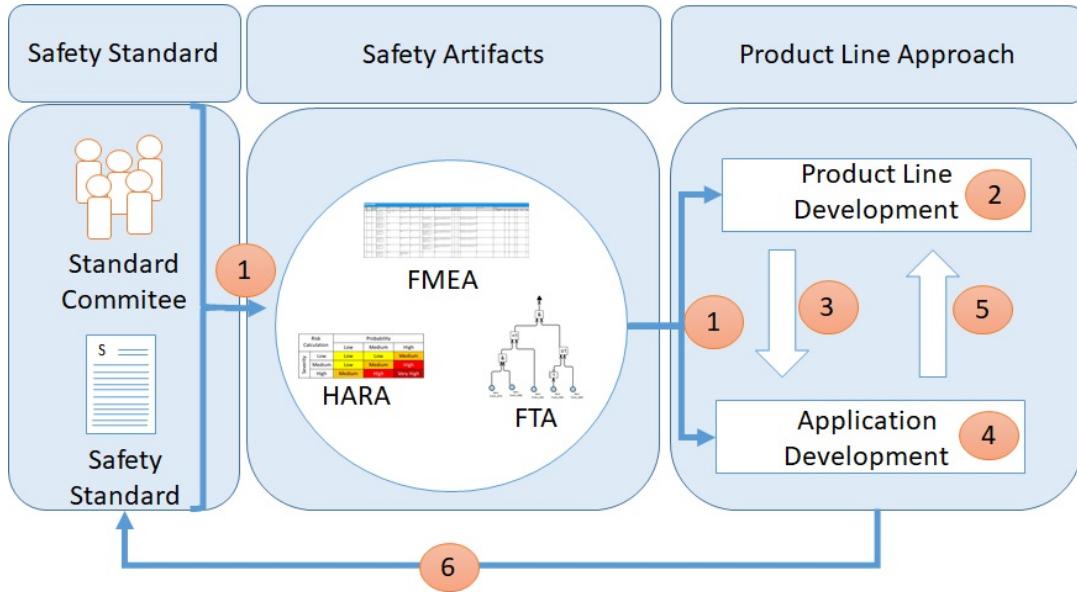
### 5.3 Integration of Product Line Artifacts to Application Development (Point 3)

The interviews show that the clone & own approach is still the most applied technique. The research question is whether the application product line techniques can be made more lightweight by tools, so practitioners benefit from product line engineering concepts. The target application development process may compensate missing qualification of reusable components by performing internal safety and security assessments. Changes from such created models need to be fed back to the reuse development, so that future application development is in sync with reuse component development. Assuming that components provide suitable safety and security artifacts, application development requires a defined way to integrate these artifacts by mapping application safety requirements to the corresponding artifacts.

### 5.4 Certifying Products (Point 4)

At the application development stage, the product line is instantiated or the components are reused.

Research may address how product lines may provide systematic approaches to build the safety at application level and lowering the certification effort for individual applications.



**Figure 1: Challenging points at integrating safety engineering to software product line development**

Integrating safety into the existing engineering processes and methodologies (i.e. Agile or DevOps) requires further research.

### 5.5 Feedback to Product Lines (Point 5)

The safety demands from application development should be communicated to the product line development. The classic product-line engineering and reuse approach assumes that the products are reused in non-safety critical domains. The usage of such products may be complicated, as safety artifacts need to be built additionally. In some cases it could be even impossible to reuse the products, if standards have certain requirements on the process, tools or programming languages. In case of open source development on product line level, the open research questions are how product line development can at least ensure, that reuse is possible in the context of safety-critical applications. Additionally, it is a practical question how companies can offer (financial) incentives or staffing to open source projects so the safety concerns are considered.

### 5.6 Long Term Impacts on Safety (Point 6)

The application of product line approaches needs to be reflected by the committees of the respective standards, so that the compliance of software product lines to standards can be guaranteed.

## 6 CONCLUSIONS & FUTURE WORK

The reuse of components in safety-critical systems is challenging, as these systems require dedicated certification or compliance with standards. In addition to safety considerations, the potential security impact has to be considered as well.

In order to determine how industry could better accommodate reuse, we explored the potential usage of product line concepts for safety artifacts and performed a study involving six companies. We want to gain more feedback, i.e. by conducting a workshop day.

All the companies we interviewed deploy some kind of reuse for their safety artifacts. Mostly the artifacts are reused by carrying them over from a similar case and adapting them after an impact analysis. Most often, worst-case scenarios and special variant cases are developed to cover all usage scenarios for generic elements. The companies differ in their opinion as to whether the existing standards provide sufficient help with reuse endeavors. All companies see improvement possibilities in the context of safety and reuse. The potential improvements mentioned most frequently are model-based approaches; introduction of simple, task-related variation management closely linked to the product variations; modularization; and better standardization.

The results give the impression that cross-domain improvement of reuse strategies in the safety context is needed. The problems in the different application domains appear to be very similar.

An open issue concerns the integration of open-source communities in safety engineering processes. Neither research nor industry currently reflects the relationship between commercial application development dealing with safety aspects and the open-source community, which produces components and systems reused in safety contexts.

In the future, the increase in systems based on artificial intelligence (AI) will lead to even more emphasis on safety considerations. As AI-based systems have no dedicated structure that can be assessed, safety artifacts cannot be derived and safety cases cannot be produced. Future research will focus on which information such black boxes need to provide and how simulation can reveal the relevant aspects required for safety certification.

## ACKNOWLEDGMENTS

We would like to thank Robert Bosch GmbH for their financial support in performing this study.

## REFERENCES

- [1] 2011. [http://www.opencoss-project.eu/sites/default/files/OPENCOS\\_Factsheet\\_V11.pdf](http://www.opencoss-project.eu/sites/default/files/OPENCOS_Factsheet_V11.pdf)
- [2] 2013. Description of pSafeCer Project (accessed: April 1st, 2019). <https://artemisia.eu/project/30-psafeacer.html>
- [3] ISO 26262. 2018. Road vehicles - Functional safety. <https://www.iso.org/standard/68384.html>
- [4] EN 50126. 2017. Railway Applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS). <https://dx.doi.org/10.31030/2759933>
- [5] EN 50128. 2012. Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems. <https://dx.doi.org/10.31030/1858892>
- [6] EN 50129. 2016. Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling. <https://dx.doi.org/10.31030/2674828>
- [7] IEC 60601-1. 2013. Medical electrical equipment - Part 1: General requirements for basic safety and essential performance. <https://dx.doi.org/10.31030/2069523>
- [8] IEC 61508. 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems. <https://www.iec.ch/functional-safety/>
- [9] IEC 62061. 2013. Safety of machinery - Functional safety of safety-related electrical, electronic and programmable electronic control systems.
- [10] Rasmus Adler, Patrik Feth, and Daniel Schneider. 2016. Safety Engineering for Autonomous Vehicles. In *DSN-W Volume*, Annual IEEE/IFIP International Conference on Dependable Systems and Networks (Ed.). IEEE, Piscataway, NJ, 200–205. <https://doi.org/10.1109/DSN-W.2016.30>
- [11] Sami Alajrami, Barbara Gallina, Irfan Sljivo, Alexander Romanovsky, and Petter Isberg. 2016. Towards Cloud-Based Enactment of Safety-Related Processes. In *Computer Safety, Reliability, and Security*, Amund Skavhaug, Jérémie Guiochet, and Friedemann Bitsch (Eds.). Springer International Publishing, Cham, 309–321.
- [12] Egbert Althammer, Erwin Schoitsch, Gerald Sonneck, Henrik Eriksson, and Jonny Vinter. 2008. Modular certification support – the DECOs concept of generic safety cases. In *6th IEEE International Conference on Industrial Informatics, 2008*. IEEE Service Center, Piscataway, NJ, 258–263. <https://doi.org/10.1109/INDIN.2008.4618105>
- [13] Katrina Attwood and Tim Kelly (Eds.). Feb 2015. *Controlled Expression for Assurance Case Development*. <http://www.opencoss-project.eu/sites/default/files/Attwood-Controlled-Expression-for-Assurance-Case-Development.pdf>
- [14] Stephan Baumgart. 2016. *Incorporating Functional Safety in Model-based Development of Product Lines*. PhD. Mälardalen University, Västerås. urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Amdh%3Adiva-31131
- [15] Markus Borg, Orlena C. Z. Gotel, and Krzysztof Wnuk. 2013. Enabling traceability reuse for impact analyses: A feasibility study in a safety context. In *International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE), 2013*, Nan Niu (Ed.). IEEE, Piscataway, NJ, 72–78. <https://doi.org/10.1109/TEFSE.2013.6620158>
- [16] Rosana T. Vaccare Braga, Onofre Trindade Junior, Kalinka Regina Castelo Branco, Luciano De Oliveira Neris, and Jaejoon Lee. 2012. Adapting a Software Product Line Engineering Process for Certifying Safety Critical Embedded Systems. In *Computer Safety, Reliability, and Security*, Frank Ortmeier and Peter Daniel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 352–363.
- [17] Alessandro Cimatti and Stefano Tonetta. 2015. Contracts-refinement proof system for component-based embedded systems. *Science of Computer Programming* 97 (2015), 333–348. <https://doi.org/10.1016/j.scico.2014.06.011>
- [18] Philipp Conmy and Iain Bate. 2014. Assuring Safety for Component Based Software Engineering. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering (HASE)*, Peter J. Clarke (Ed.). IEEE, Piscataway, NJ, 121–128. <https://doi.org/10.1109/HASE.2014.25>
- [19] Jose Luis de La Vara, Markus Borg, Krzysztof Wnuk, and Leon Moonen. 2016. An Industrial Survey of Safety Evidence Change Impact Analysis Practice. *IEEE Transactions on Software Engineering* 42, 12 (2016), 1095–1117. <https://doi.org/10.1109/TSE.2016.2553032>
- [20] Jose Luis de La Vara, Gonzalo Génova, Jose María Álvarez-Rodríguez, and Juan Llorens. 2017. An analysis of safety evidence management with the Structured Assurance Case Metamodel. *Computer Standards & Interfaces* 50 (2017), 179–198. <https://doi.org/10.1016/j.csi.2016.10.002>
- [21] J. L. de La Vara, A. Ruiz, and H. Espinoza. 2018. Recent advances towards the industrial application of model-driven engineering for assurance of safety-critical systems. *MODELSWARD 2018 - Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development 2018-January* (2018). <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85052015962&partnerID=40&md5=3f9e2347e09565bc210f5cdd1aad0f29>
- [22] André Luiz de Oliveira, Rosana T. V. Braga, Paulo C. Masiero, Yiannis Papadopoulos, Ibrahim Habli, and Tim Kelly. 2018. Variability Management in Safety-Critical Software Product Line Engineering. In *New Opportunities for Software Reuse*, Rafael Capilla, Barbara Gallina, and Carlos Cetina (Eds.). Springer International Publishing, Cham, 3–22.
- [23] Josh Dehlinger and Robyn R. Lutz. 2009. Evaluating the Reusability of Product-Line Software Fault Tree Analysis Assets for a Safety-Critical System. In *Formal Foundations of Reuse and Domain Engineering*, Stephen H. Edwards and Gregory Kulczycki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 160–169.
- [24] Ewen Denney, Ganesh Pai, and Iain Whiteside. 2015. Formal Foundations for Hierarchical Safety Cases. In *IEEE 16th International Symposium on High-Assurance Systems Engineering*, International Symposium on High-Assurance Systems Engineering, Raymond A. Paul, and Jie Xu (Eds.). IEEE, Piscataway, NJ, 52–59. <https://doi.org/10.1109/HASE.2015.17>
- [25] Dominik Domis, Rasmus Adler, and Martin Becker. 2015. Integrating variability and safety analysis models using commercial UML-based tools. In *Proceedings of the 19th International Conference on Software Product Line*, Douglas C. Schmidt (Ed.). ACM, New York, NY, 225–234. <https://doi.org/10.1145/2791060.2791088>
- [26] Dominik Domis and Mario Trapp. 2009. Component-Based Abstraction in Fault Tree Analysis. In *Computer Safety, Reliability, and Security*, Bettina Butth, Gerd Rabe, and Till Seyfarth (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 297–310.
- [27] ED-12C. 2012. Software considerations in airborne systems and equipment certification.
- [28] Huascar Espinoza, Alejandra Ruiz, Mehrdad Sabetzadeh, and Paolo Panaroni. 2011. Challenges for an Open and Evolutionary Approach to Safety Assurance and Certification of Safety-Critical Systems. In *Proceedings 2011 First International Workshop on Software Certification*. IEEE Computer Society, Los Alamitos, Calif., 1–6. <https://doi.org/10.1109/WoSoCER.2011.15>
- [29] Barbara Gallina. 2014. A Model-Driven Safety Certification Method for Process Compliance. In *IEEE 25th International Symposium on Software Reliability Engineering Workshops*. Conference Publishing Services, IEEE Computer Society, Los Alamitos, California, 204–209. <https://doi.org/10.1109/ISSREW.2014.30>
- [30] Barbara Gallina. 2015. Towards Enabling Reuse in the Context of Safety-critical Product Lines. In *Proceedings of the Fifth International Workshop on Product Line Approaches in Software Engineering (PLEASE '15)*. IEEE Press, Piscataway, NJ, USA, 15–18. <http://dl.acm.org/citation.cfm?id=2820656.2820663>
- [31] Barbara Gallina, Shaghayegh Kashiyarandi, Helmut Martin, and Robert Bramberger. 2014. Modeling a Safety- and Automotive-Oriented Process Line to Enable Reuse and Flexible Process Derivation. In *IEEE 38th Annual Computers, Software and Applications Conference Workshops*, Carl K. Chang (Ed.). Conference Publishing Services, IEEE Computer Society, Los Alamitos, California and Washington and Tokyo, 504–509. <https://doi.org/10.1109/COMPSACW.2014.84>
- [32] Barbara Gallina, Shaghayegh Kashiyarandi, Karlheinz Zugsbratl, and Arjan Geven. 2014. Enabling Cross-Domain Reuse of Tool Qualification Certification Artefacts. In *Computer Safety, Reliability, and Security*, Andrea Bondavalli, Andrea Ceccarelli, and Frank Ortmeier (Eds.). Springer International Publishing, Cham, 255–266.
- [33] Patrick Graydon and Iain Bate. 2014. The Nature and Content of Safety Contracts: Challenges and Suggestions for a Way Forward. In *The 20th IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE Computer Society, Los Alamitos, Calif., 135–144. <https://doi.org/10.1109/PRDC.2014.24>
- [34] The Assurance Case Working Group. 2018. Goal Structuring Notation Community Standard (Version 2). <https://www.goalstructuringnotation.info/>
- [35] Ibrahim Habli and Tim Kelly. 2010. A Safety Case Approach to Assuring Configurable Architectures of Safety-Critical Product Lines. In *Architecting Critical Systems*, Holger Giese (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 142–160.
- [36] Zaijun Hu and Carlos G. Bilich. 2009. Experience with Establishment of Reusable and Certifiable Safety Lifecycle Model within ABB. In *Computer Safety, Reliability, and Security*, Bettina Butth, Gerd Rabe, and Till Seyfarth (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 132–144.
- [37] Stuart Hutchesson and John McDermid. 2013. Trusted Product Lines. *Information and Software Technology* 55, 3 (2013), 525–540. <https://doi.org/10.1016/j.infsof.2012.06.005>
- [38] Omar Jaradat, Irfan Sljivo, Ibrahim Habli, and Richard Hawkins. 2017. Challenges of Safety Assurance for Industry 4.0. In *2017 13th European Dependable Computing Conference, EDCC* (Ed.). IEEE, Piscataway, NJ, 103–106. <https://doi.org/10.1109/EDCC.2017.21>
- [39] Muhammad Atif Javed and Barbara Gallina. 2018. Safety-oriented process line engineering via seamless integration between EPF composer and BVR tool. In *Proceedings of the 22nd International Conference on Systems and Software Product Line - SPLC '18 - Volume 2*, Philippe Collet, Jianmei Guo, Jabier Martinez, Christoph Seidl, Julia Rubin, Oscar Diaz, Mukelabai Mukelabai, and Thorsten Berger (Eds.). ACM Press, New York, New York, USA, 23–28. <https://doi.org/10.1145/3236405.3236406>
- [40] Michael Kaessmeyer, David Santiago Velasco Moncada, and Markus Schurius. 2015. Evaluation of a Systematic Approach in Variant Management for Safety-Critical Systems Development. In *Proceedings, IEEE/IFIP 13th International Conference on Embedded and Ubiquitous Computing*, Eli Bozorgzadeh (Ed.). IEEE Computer Society, Conference Publishing Services, Los Alamitos, California, 35–43. <https://doi.org/10.1109/EUC.2015.12>

- [41] Bernhard Kaiser, Peter Liggesmeyer, and Oliver Mäckel. 2003. A New Component Concept for Fault Trees. In *Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software - Volume 33 (SCS '03)*. Australian Computer Society, Inc, Darlinghurst, Australia, Australia, 37–46. <http://dl.acm.org/citation.cfm?id=1082051.1082054>
- [42] Jacob Krüger, Sebastian Nielebock, Sebastian Krieter, Christian Diedrich, Thomas Leich, Gunter Saake, Sebastian Zug, and Frank Ortmeyer. 2017. Beyond Software Product Lines. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*, Myra Cohen (Ed.). ACM, New York, NY, 237–241. <https://doi.org/10.1145/3106195.3106217>
- [43] Rikard Land, Mikael Åkerblom, and Jan Carlson. 2012. Efficient Software Component Reuse in Safety-Critical Systems – An Empirical Study. In *Computer Safety, Reliability, and Security*, Frank Ortmeyer and Peter Daniel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–399.
- [44] Chung-Ling Lin and Wuwei Shen. 2015. Applying Safety Case Pattern to Generate Assurance Cases for Safety-Critical Systems. In *IEEE 16th International Symposium on High-Assurance Systems Engineering, International Symposium on High-Assurance Systems Engineering*, Raymond A. Paul, and Jie Xu (Eds.). IEEE, Piscataway, NJ, 255–262. <https://doi.org/10.1109/HASE.2015.44>
- [45] John MacGregor and Simon Burton. 2018. Challenges in Assuring Highly Complex, High Volume Safety-Critical Software. In *Computer Safety, Reliability, and Security*, Barbara Gallina, Amund Skavhaug, Erwin Schoitsch, and Friedemann Bitsch (Eds.). Springer International Publishing, Cham, 252–264.
- [46] Helmut Martin, Stephan Baumgart, Andrea Leitner, and Daniel Watzenig. 2014. Challenges for Reuse in a Safety-Critical Context: A State-of-Practice Study. In *SAE Technical Paper Series (SAE Technical Paper Series)*. SAE International 400 Commonwealth Drive, Warrendale, PA, United States.
- [47] Helmut Martin, Martin Krammer, Robert Bramberger, and Eric Armengaud. 2011. Process- and Product-based Lines of Argument for Automotive Safety Cases. [http://www.artemis-emc2.eu/fileadmin/user\\_upload/Publications/2016\\_EMCA\\_Summit\\_Wien/03RMartinKrammerBrambergerArmengaudAutomotive\\_Safety\\_Case.pdf](http://www.artemis-emc2.eu/fileadmin/user_upload/Publications/2016_EMCA_Summit_Wien/03RMartinKrammerBrambergerArmengaudAutomotive_Safety_Case.pdf)
- [48] Markus Oertel, Michael Schulze, and Thomas Peikenkamp. 2014. Reusing a Functional Safety Concept in Variable System Architectures. <https://pdfs.semanticscholar.org/0e1f/6f73d05b497549d56582180e3544b6c61b3a.pdf>
- [49] Andre L. de Oliveira, Rosana T.V. Braga, Paulo C. Masiero, Yannis Papadopoulos, Ibrahim Habli, and Tim Kelly. 2014. A Model-Based Approach to Support the Automatic Safety Analysis of Multiple Product Line Products. In *Brazilian Symposium on Computing Systems Engineering (SBESC), 2014*. IEEE, Piscataway, NJ, 7–12. <https://doi.org/10.1109/SBESC.2014.20>
- [50] Sam Procter, John Hatcliff, Sandy Weininger, and Anura Fernando. 2015. Error Type Refinement for Assurance of Families of Platform-Based Systems. In *Computer Safety, Reliability, and Security*, Floor Koornneef and Coen van Gulijk (Eds.). Springer International Publishing, Cham, 95–106.
- [51] Alejandra Ruiz, Garazi Juez, Huáscar Espinoza, Jose Luis de la Vara, and Xabier Larrucea. 2017. Reuse of safety certification artefacts across standards and domains: A systematic approach. *Reliability Engineering & System Safety* 158 (2017), 153–171. <https://doi.org/10.1016/j.ress.2016.08.017>
- [52] Alejandra Ruiz, Xabier Larrucea, Huáscar Espinoza, Franck Aime, and Cyril Marchand. 2015. An Industrial Experience in Cross Domain Assurance Projects. In *Systems, Software and Services Process Improvement*, Rory V. O'Connor, Mariy Umay Akkaya, Kerem Kemaneci, Murat Yilmaz, Alexander Poth, and Richard Messnarz (Eds.). Springer International Publishing, Cham, 29–38.
- [53] Alejandra Ruiz, Alberto Melzi, and Tim Kelly. 2015. Systematic Application of ISO 26262 on a SEooC - Support by Applying a Systematic Reuse Approach. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. EDAA, [Place of publication not identified], 393–396. <https://doi.org/10.7873/DATE.2015.0177>
- [54] John Rushby and Paul S. Miner. 2002. Modular Certification.
- [55] Michael Schulze, Jan Mauersberger, and Danilo Beuche. 2013. Functional safety and variability. In *Proceedings of the 17th International Software Product Line Conference (ICPS)*, Stan Jarzabek, Stefania Gnesi, and Tomoji Kishi (Eds.). ACM, New York, NY, 236. <https://doi.org/10.1145/2491627.2491654>
- [56] Christoph Seidl, Ina Schaefer, and Uwe Abmann. 2013. Variability-aware safety analysis using delta component fault diagrams. In *Proceedings of the 17th International Software Product Line Conference co-located workshops (ICPS: ACM international conference proceeding series)*, Unknown (Ed.). ACM, New York, NY, 2. <https://doi.org/10.1145/2499777.2500721>
- [57] David J. Smith and Kenneth G. L. Simpson. 2016. *The safety critical systems handbook: A Straightforward guide to functional safety, IEC 61508 (2010 edition) & related guidance : incluiding machinery and other industrial sectors* (fourth edition ed.). Butterworth-Heinemann, Oxford, United Kingdom, Cambridge, MA. <http://proquest.tech.safaribooksonline.de/9780081008973>
- [58] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (2014), 1–45. <https://doi.org/10.1145/2580950>
- [59] Mario Trapp, Daniel Schneider, and Peter Liggesmeyer. 2013. A Safety Roadmap to Cyber-Physical Systems. In *Perspectives on the Future of Software Engineering: Essays in Honor of Dieter Rombach*, Jürgen Münch and Klaus Schmid (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 81–94. [https://doi.org/10.1007/978-3-642-37395-4\\_6](https://doi.org/10.1007/978-3-642-37395-4_6)
- [60] Dimitri van Landuyt, Steven op de beeck, Aram Hovsepyan, Sam Michiels, Wouter Joosen, Sven Meynckens, Gjalt de Jong, Olivier Barais, and Mathieu Achér. 2014. Towards managing variability in the safety design of an automotive hall effect sensor. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, Stefania Gnesi (Ed.). ACM, New York, NY, 304–309. <https://doi.org/10.1145/2648511.2648546>
- [61] Sebastian Voss, Bernhard Schätz, Maged Khalil, and Carmen Carlan. 2013. Towards Modular Certification using Integrated Model-Based Safety Cases. (2013).
- [62] Bobbi Young, Judd Cheatwood, Todd Peterson, Rick Flores, and Paul Clements. 2017. Product Line Engineering Meets Model Based Engineering in the Defense and Automotive Industries. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*, Myra Cohen (Ed.). ACM, New York, NY, 175–179. <https://doi.org/10.1145/3106195.3106220>

# How Domain-Specific Modeling Languages Address Variability in Product Line Development: Investigation of 23 Cases

Juha-Pekka Tolvanen

MetaCase

Jyväskylä, Finland

jpt@metacase.com

Steven Kelly

MetaCase

Jyväskylä, Finland

stevek@metacase.com

## ABSTRACT

Domain-Specific Modeling raises the level of abstraction beyond programming by specifying the solution directly with domain concepts. Within product lines domain-specific approaches are applied to specify variability and then generate final products together with commonality. Such automated product derivation is possible because both the modeling language and generator are made for a particular product line – often inside a single company. In this paper we examine which kinds of reuse and product line approaches are applied in industry with domain-specific modeling. Our work is based on empirical analysis of 23 cases and the languages and models created there. The analysis reveals a wide variety and some commonalities in the size of languages and in the ways they apply reuse and product line approaches.

## CCS CONCEPTS

• Software and its engineering → Software product lines; Model-driven software engineering; Domain specific languages; Abstraction, modeling and modularity; System modeling languages; feature modeling.

## KEYWORDS

Domain-specific language, domain-specific modeling, product line variability, product derivation, code generation

### ACM Reference Format:

Juha-Pekka Tolvanen and Steven Kelly. 2019. How Domain-Specific Modeling Languages Address Variability in Product Line Development: Investigation of 23 Cases. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3336294.3336316>

## 1 INTRODUCTION

Domain-specific languages and models raise the level of abstraction beyond programming by specifying the solution directly with domain concepts. This is particularly suitable in product line development, as variability and related rules are part of the language used to create specifications of variants. The final products are then generated from these high-level specifications. This automation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336316>

is possible because both the language and generators need fit the requirements of only this product line. A language-based approach for product lines manages a larger variability space, enables more freedom and flexibility and allows creating variations not possible with other approaches like parameter tables or feature models [4].

While Domain-Specific Modeling (DSM) solutions are widely applied [17, 19] and industrial cases using them are also included in the Product Line Hall of Fame<sup>1</sup>, there are few studies analyzing which kind of languages have been created and applied in industry. Studies analyzing languages within product lines tend to focus on feature modeling languages, their extensions [3, 18, 25] and comparisons of them [1, 5, 22]. When domain-specific languages are addressed, the evaluation and comparison is focused on considering different languages structures for a given situation and product line (e.g. [13, 15]). Studies analyzing larger numbers of domain-specific languages have focused on approaches to finding language constructs [19] or identification of worst practices [9].

In this study we investigate which kinds of languages companies have created for their product lines. In particular we focus on how these DSM languages define variants and reuse these variant definitions. We take an empirical approach, analyzing 23 industry cases and the DSM languages created in them. Our analysis investigates the language definitions (metamodels), their target output (e.g. program code generated), the people involved in language creation, and the reuse enabled by the language both outside of and within the models. The analysis shows that the sizes of the languages vary greatly (the largest being 14 times larger than the smallest) and that the cases use a wide variety of language structures for managing variation.

In the next section we describe the product line cases and how the languages were analyzed. Section 3 describes and gives examples of the different approaches (modeling language structures and processes) for supporting product line development that we identified. Section 4 analyzes the data and evaluates this categorization of approaches, and Section 5 summarizes our conclusions.

## 2 ABOUT THE ANALYZED PRODUCT LINES

This study is based on analyzing 23 industry cases of domain-specific modeling applied to product lines. They were selected from cases over the last 15 years where the authors have had access to the language definition – often in a consultant role supporting language and generator creation, but not having the sole responsibility for their creation. All modeling languages and generators were thus implemented in MetaEdit+ [7, 12]. The language patterns recognized, however, are not limited to any particular tool. To avoid

<sup>1</sup><http://splc.net/hall-of-fame/>

**Table 1: Product line DSM cases analyzed**

#	Domain	Targets	By	Size	Use	Approach
1	Consumer electronics	C, HTML, Docs	1	3	1	
2	Industrial automation	PLC, GUI, DB schema, net config, deploy	1	165	5	1
3	Enterprise applications	C#, DB schema	1	6	1	
4	Railway signaling	Simulation, XML	1	291	6	1
5	Signal Processing Systems	Matlab, simulation, XML	1	4	1	
6	Oil drilling	Cost calculation, documentation	1	3	1	
7	Big data applications	Java, JSON, CQL, SPARQL, SQL	2	397	4	1
8	Printing process	Ruby, XML, Docs	3	55	4	1
9	System performance	Gherkin, HTML, Docs	1	145	3	1
10	Consumer electronics	JSON	3	72	2	2
11	Telecom service	XML	1	61	2	2
12	Medical	XML, audit documents, change history	1	63	1	2
13	High-level synthesis	System C	1	450	3	2
14	Radio network	TTNC-3, simulation/animation	1	5	2	
15	An automotive system	System specification	3	62	3	2
16	Database applications	Java	3	46	1	2
17	Consumer electronics	C, localization, docs	1	403	6	3
18	Automotive architecture	Simulink, ISO26262 documents, AUTOSAR	3	652	6	3
19	Telecom	C, build automation	2	109	3	3
20	Insurance	Cobol, DB schema	3	234	6	4
21	Aerospace	C#, XSD, JSON, API	2	121	1	5
22	Automotive ECU	Python, JSON, Test document, change history	3	64	5	5
23	Software testing	Proprietary format of state machines	3	317	6	6

repeating ourselves, we chose only cases not already covered in our 2005 article [19].

All the language definitions were created freely by metamodeling, rather than being limited to being customizations, extensions or profiles of existing languages. Complete freedom was thus available when defining the language, and tooling did not restrict the language structures applied or created. Other approaches have also been seen in the literature on modeling and product lines: e.g. already available support for feature modeling could be applied directly or be extended in a preferred way, as suggested in [1, 5, 22]. Similarly, UML could be extended with stereotypes and profiles, or companies could consider defining both metamodel-based and profile-based as in [13].

The cases were chosen to cover different kinds of product lines from various industries: from consumer electronics through database applications to automotive and industrial automation systems. Table 1 summarizes these cases by their problem **Domain** and other features, in a roughly increasing order of model-level focus on product line aspects.

In all cases, DSM was applied not simply for planning, design or to support communication, but to automate development within a product line by performing model checking and generating expected output from the models: typically code but in some cases also other formal models, configurations, audit specifications and documentation. The main outputs generated are listed in Table 1 column ‘**Targets**’.

Languages were created with varying amounts of domain and language creation experience, as shown in Table 1 column ‘**By**’:

- (1) The organization in-house
- (2) An external consultant with language creation expertise
- (3) Both

The size of the DSM languages varied significantly: the smallest consisting of 46 language constructs and the largest 652 constructs. These values (where known) are shown in Table 1 column ‘**Size**’. As a comparison, UML 2.5 [14] has 247 constructs when implemented with the same GOPPRR meta-metamodel [7, 12].

The cases also varied in how far along the adoption path the language had progressed (shown in Table 1 column ‘**Use**’):

- (1) Language still under definition
- (2) Sample models made with stable language
- (3) Significant modeling of real cases as test
- (4) Real pilot project
- (5) Production use
- (6) Long term production use

Data on the use of the languages, such as how many variants within the product line had been developed, was not available for all cases. Among the cases where it was known, there was a wide variation: from just a few to hundreds of variants. Where data on the effort for creating and maintaining the languages in questions was available, it has been published in [21], and cross-tool comparisons of effort also exist, e.g. [11]. For details of the success and impact of using DSM in practice see [20, 24].

The final column in Table 1, ‘**Approach**’, is described in the following section.

### 3 HOW LANGUAGES ADDRESSED VARIABILITY

The classification of the approaches for specifying variation was gathered mostly from the language definitions available to authors. The language definitions could thus be investigated directly in MetaEdit+, and sample variant models created. For the investigation we also analyzed the variant models created, and verified our understanding from the consultants or in-house developers who were involved in creating the languages and generators.

Common to all languages was that they described the variability and left most of the commonality to existing legacy, platforms, components etc. Often a domain framework is created alongside the language, extracting further commonality. Those commonality parts are then integrated via generators.

The analysis of the cases and their languages led us to identify six approaches for addressing variability and reuse. The categorization is mostly based on language definitions but also extends into questions of process, and some approaches may require particular tool support features. The main criteria for the identification were if the created models are reused among variants, and how that reuse was established and maintained. If certain commonly reused parts of variants were maintained centrally and provided to all modelers when creating a new variant, these were considered ‘core’ models.

Data from the cases indicated the following spectrum and categorization of ‘Approach’ (the final column in Table 1):

- (1) Each model and its elements are for a single variant
- (2) Reuse of models or model elements across multiple variants
- (3) Mark/filter/modify reused models or elements for variants
- (4) Core models and variant models
- (5) Core models and languages for restricted variation of core
- (6) Multilevel: model elements become language elements

This list of approaches is not complete but reveals the typical approaches applied, some more common than others. The ordering follows our interpretation of how great a focus the approaches place on explicit product line support. In our experience the later approaches also often contain earlier ones, and as a language is developed and used, it may move further along the scale of approaches. In the following subsections we describe each approach in more detail and illustrate their use with practical examples (using real examples from cases where allowed).

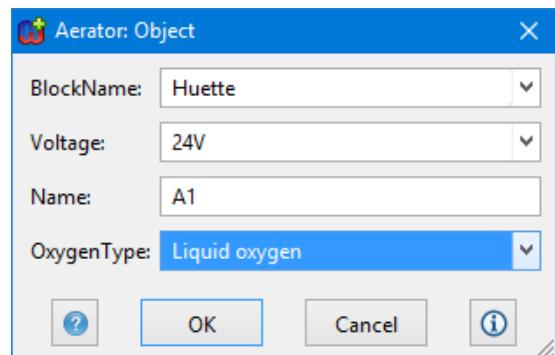
#### 3.1 Each model and its elements are for a single variant

The classical approach for applying domain-specific languages is described by [23]: The language focuses on describing the varying parts, whereas the common parts, defined in the framework, components etc., are outside the scope of the language. During the variant derivation phase the generator reads the models and integrates the variant design with the common parts in legacy code, frameworks etc.

A key feature of languages following this approach is that modelers focus on developing one variant at once: all changes made to the models are done within the context of a single variant. This approach is organizationally clear: the product development team owns all aspects of variability, has sole responsibility on testing,

versioning etc. Such languages are typical in cases such as providing telecom services per operator, an industrial automation system per factory or a service per customer.

Fig. 1 and Fig. 2 illustrate this with a practical example [16]. A company manufacturing automation systems for fish farms creates a separate specification for each customer and their system. For every Aerator being delivered as a part of the system, the language allows the modeler to set specific values for the variant (or use defaults), e.g. for the voltage and what kind of oxygen is used. The language sets this variation space and with a simple example like in Fig. 1, this part of the language can be considered to be a configurator over the parameters of the whole product line offering.



**Figure 1: Setting values for a variant of Aerator**

In practice, all variation is not so simple. The unit of variability can be any aspect of the product line that can be expressed with a language. For instance, additional variation related to an Aerator can be its location, relation to fish ponds, network and power supply etc. DSM languages enable capturing this richer variation too, as illustrated in Fig. 2, which shows a portion of an automation system in which a particular Aerator is one varying part. For example, the blue ellipses are ponds for the fishes and one aspect of variability is their location. The variant also specifies other customer-specific features (lights, feeding, monitoring Ph level, muddiness, temperature etc.) and a network with related configuration. Location, order, connection and behavior are all kinds of variation that can be difficult or impossible to express with wizards, parameter tables or feature models, but fit well to DSM.

From the variant models like Fig. 2 the company produces PLC code for an automation system, database definitions for storing persistent data, network configuration for the given setup, installation guidelines, material needs etc. The location of the ponds is used to produce a user interface to monitor and control the automation system.

Languages focusing on a single variant are useful when variant development teams work independently and there is no need to share functionality created for one variant with others. If such a need arises without proper language support, often the only choice left is clone and own – and keeping track of changes made in the original variant model and copies.

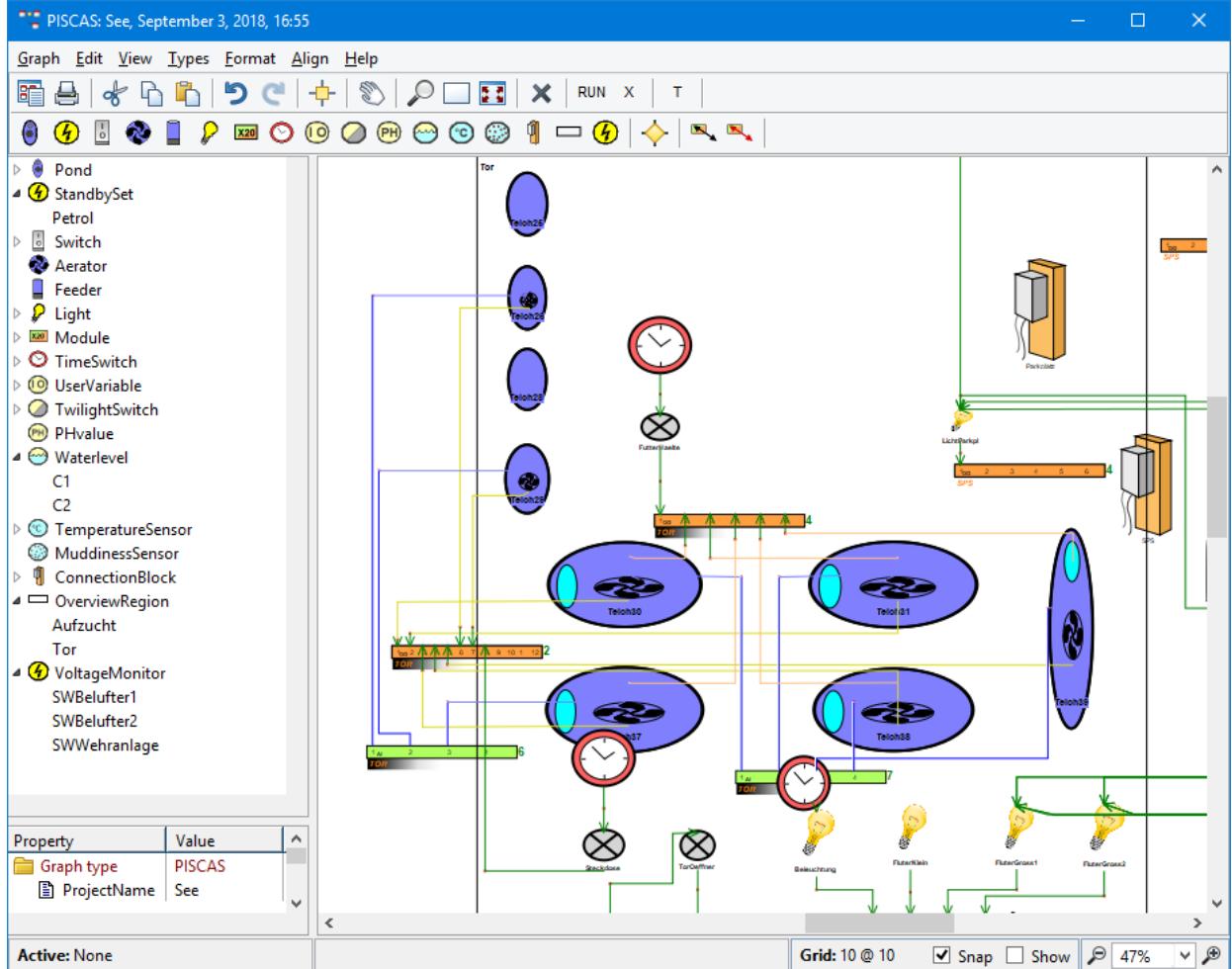


Figure 2: Variant of fish farm automation for a given customer

### 3.2 Reuse of models or model elements in multiple variants

When more variants are being developed, often a need arises to reuse existing work: Functionality that is already defined for one variant is found useful for others too. To avoid clone and own, a sub-model or element is allowed to be referred to by more than one variant. (Having a hierarchy of models, and even allowing reuse of a sub-model, are also found in Approach 1, but there the reuse is always within a single variant.) As in Approach 1, the top-level model's name is often effectively the variant name.

Fig. 3 and Fig. 4 illustrate examples of reuse in the CPL language used to specify how calls are processed for a telecom operator's customers. Fig. 3 specifies one customer's call redirection service which uses voicemail redirection (subaction block at the bottom) based on location. Similarly, another customer's service for location-based redirection defined in Fig. 4 reuses the same voicemail redirection service in a more complex rule. The reused service is defined only once and applied in several variants. The variant models then detail

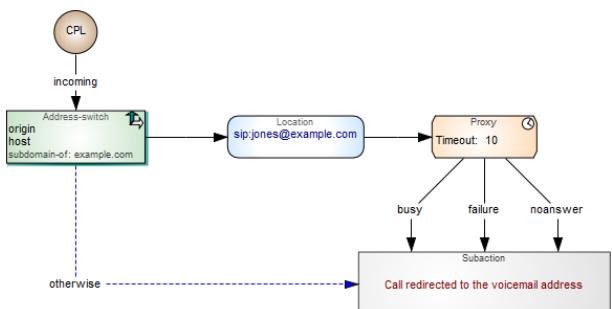
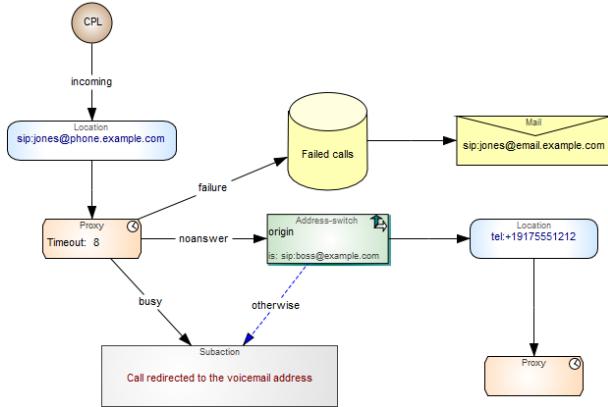


Figure 3: Location-based call redirection to voice mail

the contexts (e.g. busy, noanswer, etc.) in which the reused subaction is used. In other words, the language knows the correct ways to reuse the subaction and can guide and check variant creation during specification.



**Figure 4: Call redirection reusing voice mail redirection**

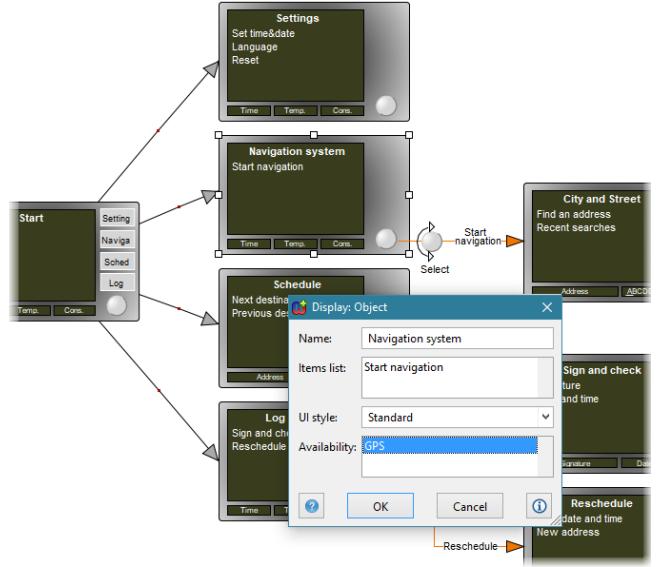
The reused service can be provided as a whitebox or a blackbox component depending on whether its details are made visible. Introducing model reuse raises similar questions as any other kind of reuse: Can reusers see the details, can they change them, is this reuse by reference or copy, etc. Also questions arise about the different lifecycles of reused parts and variant-specific parts: Are updates and bug fixes to reused parts delivered automatically to their users? Modeling tools may also offer support for governance and management with access rights, review policy etc. We discuss the role of tools for variability later in Section 3.4.

While this small example is based on the same language, i.e. reusable services are specified with the same language as all other services, it is possible to have different languages for different users (e.g. one for service creators at operator side and another for implementing the services, as presented in [6]). When languages are different among teams this also clarifies the responsibility of core team and variant development teams.

### 3.3 Explicit ability to mark, filter or modify models or elements for variants

Once elements can be reused as in Approach 2, there often arises a need to change those elements somewhat in particular variants. For example, model elements can be marked as only being present or active in certain variants (or other conditions), or a higher-level model can specify to include a lower-level model with certain filtering or configuration. The former can be considered bottom-up variant specification, and the latter top-down. In both cases, new concepts to explicitly specify variation are added to the language (in contrast to Approach 2, where the language remained the same.)

An example of bottom-up variant specification is seen in Fig. 5, which shows an excerpt of the UI and interaction flow of a car infotainment system. A definition for display 'Navigation system' is selected and its property dialog is shown. In the dialog, the Availability property value defines that this 'Navigation system' display is provided only when the GPS module is available. Also all other functionality related to the Navigation display like its menus, knobs etc. are excluded as well as any other displays reachable only via the Navigation display. While the variability is set here based



**Figure 5: Variation of infotainment system for GPS sensor**

on the sensors available, it could be also based on product names (e.g. available only for infotainment system 'CarInfo3000'), features (e.g. available only for Autopilot) etc. A downside of this approach is that the properties like availability need to be entered – and later possibly modified – in multiple places (e.g. displays in this example).

Rather than adding variant information to individual language elements, the top-down approach extracts variant information to a new, higher model layer: A dedicated language is used to configure existing models defined for variants. Fig. 6 illustrates this top-down approach, showing a watch product family consisting of three members: Delicia, Ace and Sporty. While here one model specifies all three variants, each product could have a separate configuration model. In this example product line, each watch contains a display (on the left) and a logical behavior (on the right) whose submodel contains a number of applications, such as alarm or stopwatch.

One application, Stopwatch, is illustrated in Fig. 7. It allows the user to start the stopwatch running, showing the elapsed time and an icon while in that Running state; stop the stopwatch; and reinitialize the elapsed time. All three variants use the same specification of the Stopwatch application, yet in a different manner as specified in the higher-level family model. For example, Sporty has an icon for Timer but none for Stopwatch, so the stopwatch icon is not shown in Sporty when Stopwatch is running.

With this kind of language, variant developers can decide if existing variant models already contain the needed functionality and add a configuration using them. New products can thus be created quickly by referencing and configuring existing functionality. Alternatively, the development team can extend an existing functionality for the new variant but in a manner that the functionality for existing products is maintained.

With languages following this approach, the variant development teams take responsibility for the whole product line, not just their variant. The models created then need to be versioned and

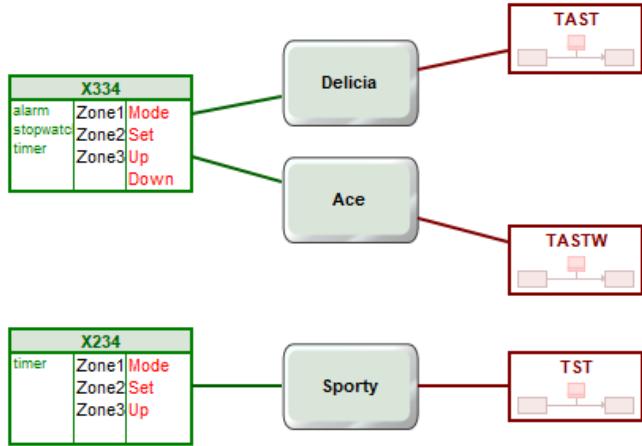


Figure 6: Watch family

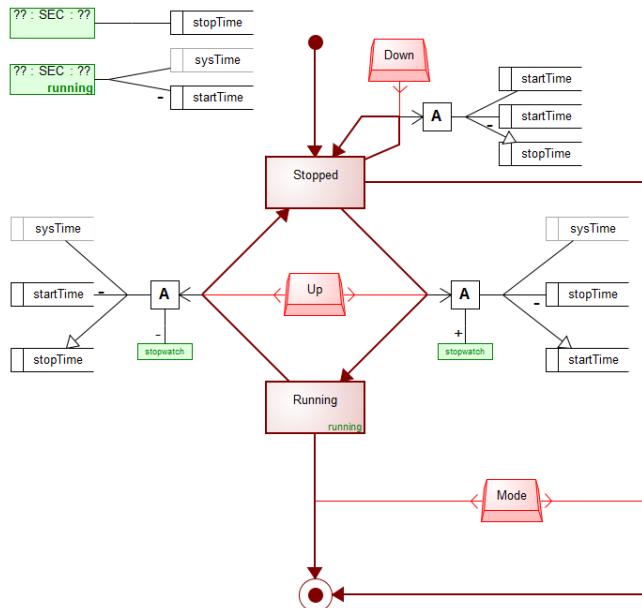


Figure 7: Stopwatch application supporting several variants (Ace, Delicia, Sporty)

tested together. This may well require establishing different policies for variant development teams than when using languages whose models each address only one variant.

### 3.4 Core models and variant models

Product line companies generally differentiate the platform and other commonalities, developed by the ‘core team’, and the individual variants developed by ‘product development teams’. While DSM has generally already moved the core commonalities out of the modeling language, modeling several variants generally reveals some parts of variant models that are first reused in another variant (as in Approach 2), and later recognized as being useful to all variants. These ‘core models’ can then be supplied to each team

starting a new variant, and can be maintained separately by a ‘core modelers’ team. This explicit recognition, decision and separation of core models is what distinguishes Approach 4 from Approach 2.

At least initially there are generally no clear differences in the domain itself on which parts can be core and which parts can be specified for a given variant, and so the modeling language can be the same for both teams. When languages do not address variation it is then left to the tooling and processes. Such tool-based approaches then vary based on tool functionality and their integration with other data and the other tools used.

Tools may provide in-built support for working with core models and variant models. For example, models can be defined in different repositories or projects within the modeling tool and their access can be restricted for different teams. Typically product development teams may not be allowed to change core models or their individual elements but just use them. For example, in a MetaEdit+ multi-user repository, a project’s model elements can be set to be read-only, so that only members of the core team can change them and others may only refer to them in their designs. When the core changes, the variant development teams see the updates automatically.

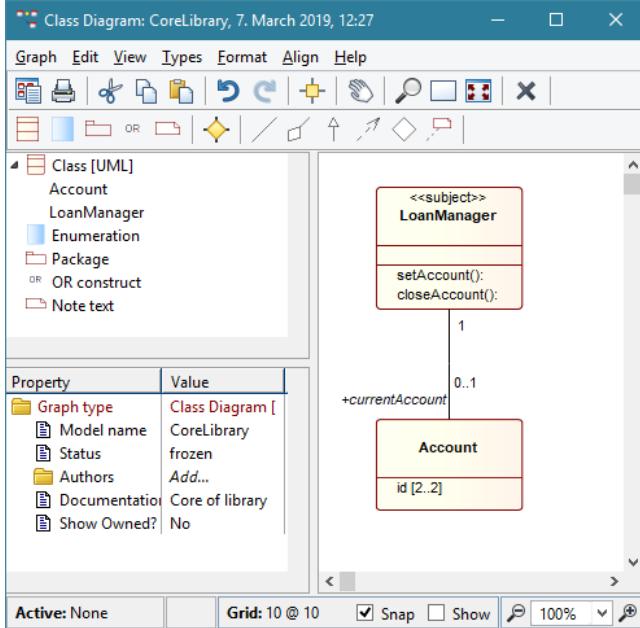
Alternatively, it is possible that core models are maintained separately and the core team can release them with dedicated content and versions. For example, different versions of the core models can be released to different variant development teams or different sets of core models can be released to different variant teams. Variant development teams may then also decide when to update to the next version of the core.

Tool support is needed to manage sharing and updating core models for teams applying them in product development. For example, in MetaEdit+, a core team can release models to be imported in separate repositories used by product development teams – and different rights can be set if changes to them are allowed or not. When the core models change, the core team can release the changed models and choose that they will automatically update previously delivered core models. This approach is applicable if there are subcontractors involved that do not get access to all designs, or development teams are working remotely in different locations, or there is otherwise a need to keep variant models made for different customers separate (as in Approach 1).

### 3.5 Core models and languages for restricted variation of core

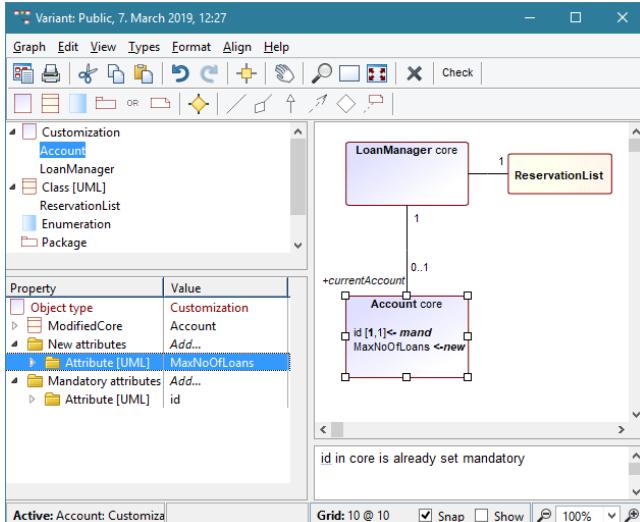
Among the cases analyzed we also found examples where modifications to the reused core were allowed but in a restricted way. The need for modification and the type of modifications allowed were identified during domain analysis and language definition. This approach is illustrated with a class diagram using an example of library variants from [2]. Fig. 8 shows a small core model in MetaEdit+ containing just two classes that are common for all library systems. This model is created by the core development team.

A variant development team then creates a product using a modeling language that restricts the modifications to the core models. Fig. 9 illustrates the use of this language. A variant of library system for public libraries uses both ‘LoanManager’ and ‘Account’ from the core, but allows adding new functionality created just for the



**Figure 8: A small example of core model for a library**

variant, like ‘ReservationList’ in the example. More importantly, the ‘Account’ class from the core has been modified in two ways – as allowed by the modeling language. A new attribute for maximum number of loans has been added and an existing attribute for identification is made mandatory. The notation of the language makes clear the parts used from the core and the type of modifications that have been made to them. Both the notation and the kinds of modifications allowed are set by the language creators.



**Figure 9: Creating a variant by customizing the core**

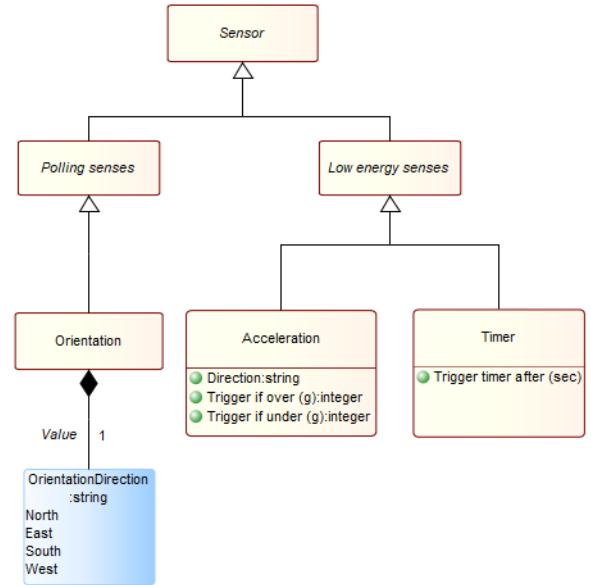
With this approach the core parts can evolve and be updated, yet at the same time their modification in variant projects is possible.

The tooling can also check for possible inconsistencies and report them as illustrated in Fig. 9. Here the variant development team has made the ‘id’ attribute mandatory but it has later been changed to be mandatory in the core too. MetaEdit+ checks the variation rules and reports on possible inconsistencies: See the error text below the diagram stating ‘id in core is already set mandatory’. Variability rules defined are not only applied when creating the models but also while editing, generating variants etc.

While this example is very small and shows just three ways to extend the core, the same principle for setting restrictions on how core models can be modified can be applied for other kind of extension rules and for other, more domain-specific, languages than the class diagrams shown here.

### 3.6 Multilevel: model elements become language elements

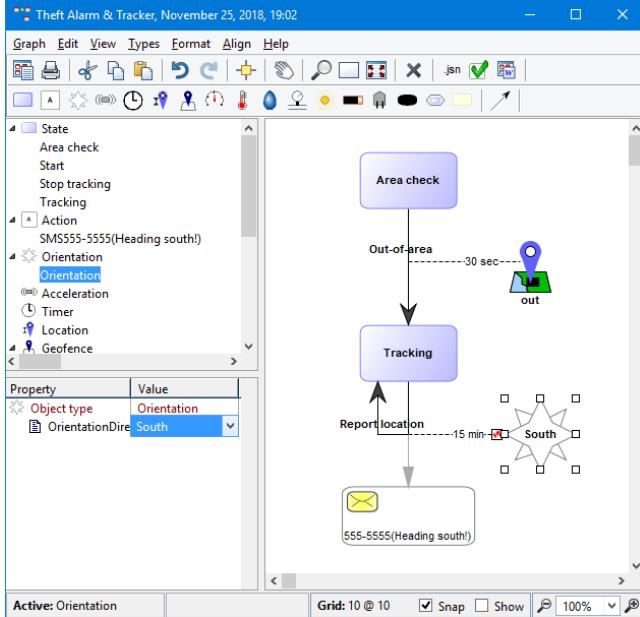
Finally, in one particular product line DSM case we identified a pattern known as multilevel modeling: A model created in one role or by certain developers is used to form language elements for the others. This is illustrated below with a small example on sensors available for a variant. Fig. 10 shows a model of possible sensors and their characteristics, such as what they measure and how they can be applied, e.g. with continuous polling or low energy usage. In the multilevel approach, a model specifies here capabilities of the device been used. For example, there is a polling sensor for orientation providing a compass direction.



**Figure 10: Adding an Orientation sensor language concept**

This model then forms a part of the language used to specify applications for the given device variant. Fig. 11 illustrates this language being used to specify tracking using an Orientation sensor. As it is polling-based, the model states that data from the Orientation sensor is read every 15 minutes. If the compass direction is

South, the bottom transition is triggered and a message is sent to the given phone number. All available language concepts – including the newly-added Orientation sensor – are visible in the toolbar of the modeling tool. Refining the sensors in Fig. 10 thus updates the language being used for variant development in Fig. 11.



**Figure 11:** Using an Orientation sensor in a variant

As in the previous two approaches, there is a clear possibility for having different kinds of users or roles. Changing the language also opens up the normal tool support questions of language evolution: fortunately one of the strengths of the tool in this case [10].

#### 4 EVALUATION OF THE CASE DATA AND PRELIMINARY ANALYSIS

Among the cases analyzed, the commonest approaches were Approach 1: languages for a single variant (9 cases) and Approach 2: languages designed for reusing models or model elements among variants (7 cases). The large number of languages focusing on single variants is understandable as they are the simplest in terms of process, and proven practices for creating such languages have long been available (e.g.[23]). The entire support for product line engineering is provided by the language and generators, with no explicit mention of variants, and no cross-variant reuse in the models.

Our experience that the Approaches form an ordering received some support and no refutation from the data. Two cases which originally focused on single variants (Approach 1) later encountered the need for reuse and moved to Approach 2 (#10, #11), and two others to Approach 3 (#17, #19).

When analyzing how the languages were developed, we identified that almost all languages developed in-house, without support from external consultants, fell into Approaches 1 and 2. A one-tailed Pearson correlation of 0.52 was found between increasing explicitness of product line approach and increasing categories of language

creator expertise involved, statistically significant with  $P < 0.01$ . No correlation (0.04) was found between language creators and how far the language progressed along the path to full production use.

It is clear from Table 1 that the sizes of the languages vary considerably: the largest is 14 times the size of the smallest. The data also showed that among the largest languages were those that have been applied for a long time (#17, #18, #20, #4). A one-tailed Pearson correlation of 0.60 was found between Size and Use, statistically significant with  $P < 0.005$ . No correlation (0.07) was found between Size and Approach.

We also investigated whether the type of modeling language chosen had an effect on Size, Use or Approach. Language types were classified based on their main focus or model of computation as Location, Costs, UI, Query, Structure, State machine, UI flow, Data flow, Action flow, Sequence and Data structure. A single case often included two language types. Although all cases using the simplest four language types at the start of this list also used the simplest single-variant Approach 1, the effect was not statistically significant because of the small sample size for those language types (only one or two cases for each of those language types).

#### 5 CONCLUSIONS

In this study we analyzed 23 industry cases where domain-specific modeling languages were created for product line development. In particular we investigated how domain-specific languages handle reuse of the variant specifications. The cases were selected to cover different kinds of product lines from various industries. The oldest languages included in the analysis have been used for over 10 years and the newest only just created.

The analysis of languages focused on their definitions (metamodels), along with broader information about the area targeted and the creation and adoption process. We applied the languages in a tool to investigate how variant specifications can be created and reused.

In the cases studied, we identified six different approaches to supporting product line development in DSM. With even the most basic DSM language already supporting product line development implicitly, it was interesting to see that over half the languages were developed to provide further product line support. A third of the languages added reuse of variant models or model elements, and another third offered various kinds of explicit modeling of variation. The list of approaches is obviously not complete, but it gives an indication of the richness of language structures that can be applied to support product lines with DSM.

The majority of these languages were created in-house, without support from external consultants. In the cases where other language definition approaches were applied external consultants were involved. This can be explained that these cases called for more experience on language design (like adding configuration for variability) or even on the particular tool, at least at the highest end (support for multi-level modeling).

More research work is needed to analyze the benefits of each approach, its costs in terms of the creation and maintenance effort, and what aspects within the case have led to the chosen approach. Future work can also extend the number of cases analyzed, and cover DSM solutions created by other tools. We are not aware

of studies investigating numerous industry cases that used other tools, and welcome such studies addressing language creation with other tools and technologies. Other research methods could also be applied, e.g. giving more detailed analysis of language evolution within an individual case. Conversely, the scope could be widened by a survey.

Perhaps the most encouraging aspect of this research was to see how many cases were able to use MetaEdit+ to successfully implement a product line approach with DSM, even with only in-house resources. With so much development still taking place by clone and own, any approaches and tools that can offer organizations a reliable path to product line development are welcome.

## REFERENCES

- [1] Acher M., Heymans P., Collet P., Quinton C., Lahire P., Merle P., Feature Model Differences. In: Ralyté J., Franch X., Brinkkemper S., Wrycza S. (eds) Advanced Information Systems Engineering. CAiSE 2012. Lecture Notes in Computer Science, vol 7328. Springer, 2012
- [2] Atkinson, C., Bunse, C., Bayer, J., Component-based Product Line Engineering with UML, Pearson Education, 2002
- [3] Cognini, R., Corradini, F., Polini, A., Re, B., Extending Feature Models to Express Variability in Business Process Models, In proceedings of Advanced Information Systems Engineering Workshops, Springer, 2015
- [4] Czarnecki, K., Eisenecker, U., Generative Programming, Methods, Tools, and Applications, Addison-Wesley, 2000
- [5] Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wąsowski, A., Cool features and tough decisions: a comparison of variability modeling approaches. In Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '12). ACM, 2012
- [6] Hulshout, A., Service Creation with MetaEdit+: A telecommunications solution. Presentation at Code Generation Conference, Cambridge, May 19th, 2007
- [7] Kelly, S., Lyytinen, K., Rossi, M., MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In: Constantopoulos P., Mylopoulos J., Vassiliou Y. (eds) Advanced Information Systems Engineering. CAiSE 1996. Lecture Notes in Computer Science, vol 1080, Springer, 1996
- [8] Kelly, S., Tolvanen, J.-P., Domain-Specific Modeling: Enabling Full Code Generation, Wiley-IEEE Computer Society Press, 2008
- [9] Kelly, S., Pohjonen, R., Worst Practices for Domain-Specific Modeling, IEEE Software, Vol. 26, 4, 2009
- [10] Kelly, S., Tolvanen, J.-P., Collaborative creation and versioning of modeling languages with MetaEdit+. In: Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS '18), Babur, Ö., Strüber, D., Abrahão, S., Burgueño, L., Gogolla, M., Greenyer, J., Kokaly, S., Kolovos, D., Mayerhofer, T., Zahedi, M. (eds.). ACM, pp. 37–41, 2018
- [11] El Kouhen, A., Dumoulin, C., Gerard, S., Boulet, P., Evaluation of Modelling Tools Adaptation. CNRS HAL hal-00706701, 2012, <http://tinyurl.com/gerard12>
- [12] MetaCase, MetaEdit+ Workbench 5.5 User's Guide, [www.metacase.com](http://www.metacase.com), 2018
- [13] Mewes, K., Domain-specific Modeling of Railway Control Systems with Integrated Verification and Validation, Ph.D. thesis, University of Bremen, 2009
- [14] Object Management Group, Unified Modeling Language, Version 2.5.1, 2017
- [15] Preschern, C., Kajtazovic, N., Kreiner, C., Evaluation of Domain Modeling Decisions for Two Identical Domain Specific Languages, Lecture Notes on Software Engineering 2, 1, 2014
- [16] Preschern, C., Leitner, A., Kreiner, C., Domain-Specific Language Architecture for Automation Systems: An Industrial Case Study, In: Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications (eds. Störzl et al.) DTU Informatics, 2012
- [17] Sprinkle, J., Mernik, M., Tolvanen, J.-P., Spinellis, D., What Kinds of Nails Need a Domain-Specific Hammer? IEEE Software, Vol. 26 , 4, 2009
- [18] Sousa, G., Rudametkin, W., Duchien, L., Extending feature models with relative cardinalities, Proceedings of the 20th International Systems and Software Product Line Conference, Beijing, China, ACM, 2016, <https://doi.org/10.1145/2934466.2934475>
- [19] Tolvanen, J.-P., Kelly, S., Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences. Proceedings of the 9th International Software Product Line Conference, Obbink, H., Pohl, K. (eds.). Springer-Verlag, LNCS 3714, 2005
- [20] Tolvanen, J.-P. and Kelly, S. Model-Driven Development Challenges and Solutions - Experiences with Domain-Specific Modelling in Industry. In Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development, SCITEPRESS Science and Technology Publications, Lda, 2016
- [21] Tolvanen, J.-P., Kelly, S., Effort Used to Create Domain-Specific Modeling Languages. In ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS 18), ACM, New York, NY, USA, 2018
- [22] Wahyudianto, Budiardjo, E., Zamzami, E., Feature Modeling and Variability Modeling Syntactic Notation Comparison and Mapping, Journal of Computer and Communications, 2014
- [23] Weiss, D., Lai, C.T.R., Software Product-line Engineering, Addison Wesley, 1999
- [24] Whittle, J., Hutchinson, J., Rouncefield, M., The State of Practice in Model-Driven Engineering, IEEE Software, 31, 3, 2014
- [25] Zaid, L., Kleinermann, F., Troyer, O., Feature Assembly: A New Feature Modeling Technique, Proceedings of 29th International Conference on Conceptual Modeling, Parsons, J., Saeki, M., Shoval, P., Woo, C., Wand, Y. (eds.). Springer, 2010

# Software Product Line Engineering: A Practical Experience

Jose-Miguel Horcas

horcas@lcc.uma.es

Univ. de Málaga, CAOSD group, Spain

Mónica Pinto

pinto@lcc.uma.es

Univ. de Málaga, CAOSD group, Spain

Lidia Fuentes

lff@lcc.uma.es

Univ. de Málaga, CAOSD group, Spain

## ABSTRACT

The lack of mature tool support is one of the main reasons that make the industry to be reluctant to adopt Software Product Line (SPL) approaches. A number of systematic literature reviews exist that identify the main characteristics offered by existing tools and the SPL phases in which they can be applied. However, these reviews do not really help to understand if those tools are offering what is really needed to apply SPLs to complex projects. These studies are mainly based on information extracted from the tool documentation or published papers. In this paper, we follow a different approach, in which we firstly identify those characteristics that are currently essential for the development of an SPL, and secondly analyze whether the tools provide or not support for those characteristics. We focus on those tools that satisfy certain selection criteria (e.g., they can be downloaded and are ready to be used). The paper presents a state of practice with the availability and usability of the existing tools for SPL, and defines different roadmaps that allow carrying out a complete SPL process with the existing tool support.

## CCS CONCEPTS

- **Software and its engineering → Software product lines; Abstraction, modeling and modularity; Software notations and tools; Software usability;** • **General and reference → Empirical studies.**

## KEYWORDS

SPL in practice, state of practice, tool support, tooling roadmap

### ACM Reference Format:

Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2019. Software Product Line Engineering: A Practical Experience. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3336294.3336304>

## 1 INTRODUCTION

An increasing number of software development companies are adopting *Software Product Line* (SPL) approaches [8, 47]. However, the field of SPL is quite broad and continuously changing [57]. Technical issues such as the development of the SPL infrastructure, including new practices, processes, and tool support require a great investment [8]. Thus, despite the amount of successful stories

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336304>

about the use of SPL engineering<sup>1</sup>, industry has not yet solved the variability and reuse management problem and continues to experiment with their own solutions and approaches [13]. This is partly due to the fact that companies are reluctant to adopt technically sound academic approaches due to the lack of mature tool support. Hence, the success of SPL depends on good *tool support* as much as on complete *SPL engineering processes* [6].

Regarding the processes, SPL approaches typically cover the *domain* and *application engineering* processes that include activities such as variability modeling and artifact implementation (domain engineering) and requirements analysis and product derivation (application engineering) [4, 11], but set aside other activities that are important for companies, such as the analysis of *non-functional properties* (NFPs) or *quality attributes* and the *evolution* of SPL's artifacts [41]. When considered, these activities are usually integrated into the traditional SPL process by reusing existing mechanisms, which were not specifically designed for that purpose (e.g., using attributes of extended variability models to model NFPs [10]). The most common is to find companies that only adopt a minor part of an SPL methodology (e.g., implementing variability with annotations) [30], which is sometime abandoned after a short period of time because of the lack of integration among the SPL activities. Only a few companies with enough financial resources have succeeded on using SPLs, due in most cases to the development of their own tools or the use of commercial tools (e.g., pure::variants [12], Gears [40]) that are not so accessible for smaller companies.

Besides, although tool support is of paramount importance for the SPL management process [6], most existing tools only cover specific phases of the SPL approach (e.g., variability modeling or artifacts implementation). Those few tools that support several phases (e.g., FeatureIDE [64]) demand to fit an implementation technique such as *Feature-Oriented Programming* (FOP) [53], *Aspect-Oriented Programming* (AOP) [38] or *annotations* [37], depend on the development IDE (e.g., Eclipse) or present some important limitations [20]. For instance, applying classical SPL approaches (e.g., FOP or AOP) to web engineering is challenging because of the nature of web applications that require the simultaneous use of several languages (JavaScript, Python, Groovy,...) in the same application [30]. In addition, there are few works specifically focused on studying the SPL tool support [6, 43, 51] and, they usually report information that is extracted from the tool documentation or reference papers without really testing the tool *availability* and *usability*.

This paper explores the existing tool support for SPL from a practical point of view, although it does not pretend to be a systematic review. The objective is to check out the existence of enough mature tool support for carrying out a complete SPL process. For each activity in the domain and application engineering, we identify the desired requirements that tools should provide to deal with complex SPL processes and application domains, and analyze the

<sup>1</sup><http://splc.net/hall-of-fame/>

possibilities and limitations of each tool. The paper answers the following Research Questions:

- RQ1:** *What are the tools that provide some kind of support for SPLs? Are they available, usable, and keep up to date?* To answer to this question, this paper presents a *state of practice* of the SPL tools, focusing on their availability and usability and selecting those that a company could use in its development process to successfully adopt an SPL process (Section 4).
- RQ2:** *How the tools behave when they are used to develop complex SPLs? (Either because a specific SPL development approach is required or because SPLs for complex application domains need to be developed.)* We answer to this question by *empirically analyzing* the most usable tools. We use a running case study based on an SPL process with demanding characteristics such as clonable features [24, 60], variable features [21, 22], attributes [10], huge feature models, and so on (Section 5).
- RQ3:** *Is it possible to carry out a complete SPL process with the existing tool support? That is, is it possible to cover all activities of complex approaches including dealing with NFPs, and managing the evolution of SPLs?* We answer to this question by defining up to 12 different *roadmaps* of tools that partially or completely support all phases of an SPL process (Section 6).

The paper is structured as follows. Section 2 discusses related work. Section 3 motivates our study showing the requirements of complex SPL approaches. Section 4 presents a state of practice of the existing tools. Section 5 analyzes the most usable tools. Section 6 defines the tool roadmaps to carry out a complete SPL. Section 7 discusses threats to validity and Section 8 concludes the paper.

## 2 RELATED WORK

Existing works have investigated the SPL processes in great detail [17, 56, 57], but has only surfaced its tool support [6, 43, 51].

### 2.1 Software Product Line processes

Multiple systematic literature reviews (SLRs) and surveys have been published covering different aspects of SPL engineering [11, 16, 17, 48, 56, 57], such as the level of alignment in the topics covered by academia and industry [11], the level of tool support [16] or the most researched topics in SPL [56, 57]. From these studies the phases and topics of SPL engineering in which academia and industry are more interested or that deserve more attention can be identified. These SLRs highlight some interesting conclusions. For instance, architecting [15] is the dominating SPL activity, covered by 38% of surveyed papers (56% of them are industry papers) [57].

However, there are other activities that are becoming important in the context of SPL with the emerging of new application domains, such as the Internet of Things (IoT) [55] or Cyber-Physical Systems [35], and that are not receiving the required attention. Examples of these activities are the optimization of large-scale variability models [48, 49], the variability modeling of quality attributes [31, 68], the management of NFPs [34, 48], and the evolution of the SPL models [32, 41]. This imposes new challenges to the existing development and analyses processes, as well as to the tool support. Even so, as exposed by those SLRs [11, 57], the variability in quality attributes is a concern only in 6% of the papers, and NFPs are discussed only in 5% of all the papers [57].

### 2.2 Software Product Line tools

Few works study the tool support for the SPL processes specifically [6, 16, 43, 51]. They are SLRs or surveys that are normally done only from the perspective of the documentation found for each tool, and the characteristics listed and discussed in that documentation. Moreover, most of the details about tools are covered in gray literature, thesis, and websites, that are not usually considered as primary studies in SLRs. Commercial tools (e.g., Gears [40] and pure::variants [12]) present the additional problem of the intellectual property protection of their technical details [6]. Concretely, in [12], a demonstration of pure::variants across the product line lifecycle is described, but it only surfaces the tool without providing further technical details. The same occurs with Gears in [40].

One of the most recent systematic studies [6] covers tools documented in research papers from 1997 until 2015 (although the study was published in 2017). This is an interesting study to know the general characteristics of these tools (e.g., technology used in its implementation, if it has a graphical or textual notation, etc.). Others similar, but older studies are [51] (published in 2014) and [43] (published in 2010). However, the most recent tools reported by these studies date from 2013 and 2005, respectively.

The conclusion is that these kinds of studies are not enough to select the most appropriate tool to provide support for an SPL process. This is basically because only information about the high-level phases covered by each tool is provided, omitting the details about the specific topics covered for each phase. In addition, the information is extracted from the tool documentation or a reference paper, and thus, these studies stay outdated very soon because, in most of the cases, they are not trying and striving directly with the tools, downloading, installing, and executing the tools or even checking their online availability – i.e., many of the tools included in existing studies are not available at all. There are even tools referenced in these papers that have never been implemented [51].

## 3 MOTIVATION AND CASE STUDY

An SPL approach should cover at least the domain engineering and the application engineering processes with their typical phases and activities (Figure 1): (1) variability and dependency modeling in the Domain Analysis (DA) phase; (2) feature selection and product configuration in the Requirements Analysis (RA) phase; (3) the development of reusable software artifacts in the Domain Implementation (DI) phase; and (4) variability resolution and product generation in the Product Derivation (PD) phase [4]. However, demanding requirements of real SPL projects expose the needs of additional activities that are present in any software engineering process and that are essentials for the successful adoption of SPLs in industry. Some examples are the analysis of NFPs or quality attributes, or the evolution of the SPL's artifacts, among others.

To illustrate those points, we present *WeaFQAs* [29, 31, 34], an example of an SPL process that extends the engineering framework for SPL [52] and demands specific needs in each of the SPL phases according to the current applications and domains where WeaFQAs may be used. In this section, we describe the particular needs that WeaFQAs imposes in each phase of the classical SPL framework and enumerate the requirements that tools should satisfy.

### 3.1 Case Study: the WeaFQAs SPL process

WeaFQAs [29, 31, 34] is an SPL process to manage the *operationalizations* of quality attributes. The operationalization of a quality attribute (e.g., security) is the association of a function (e.g., the encryption of a message) to a goal (e.g., providing security). WeaFQAs introduces the concept of *Functional Quality Attribute* (FQA) as the specific functionality that is incorporated into the applications to fulfill the desired quality attribute. WeaFQAs promotes the variability modeling and customization of FQAs separately from the applications, and their later incorporation into them, exploiting the benefits of SPLs (e.g., reusability, adaptability,...). WeaFQAs follows the classic framework for SPL engineering [52] (Figure 1) – i.e., DA, RA, DI, and PD., but needs to extend it to take into account the specific problematic of FQAs, which requires additional activities in each of those phases (see activities in bold in Figure 1).

### 3.2 Requirements of complex SPLs

**Domain Analysis (DA).** Modeling a family of FQAs starts with the characterization of the FQAs and their variability modeling (Figure 2). For each quality attribute (e.g., security) the FQAs that are required to satisfy it are modeled (e.g. encryption, authentication). For instance, characterizing the encryption FQA includes the identification of the different encryption algorithms, the available security frameworks that implement it, along with their variables and parameters such as the key length, block cipher mode of operation, and padding; the dependency relationships between encryption and others FQAs (e.g., hashing); as well as the usage context [34].

Nevertheless, modeling the FQAs variability is more complex than only considering their high degree of variability. Since each FQA (e.g., encryption) can be applied in several points of the application with different configurations (e.g., RSA or AES algorithms), each FQA needs to be modeled as a *clonable feature* [24, 60] (see Encryption[1..\*] in Figure 2). Moreover, some FQAs include variables that require to provide specific values (e.g., 512 kB as the average size of the messages to be encrypted). This requires to model them as *variable features* [21, 22] (see MessageSize in Figure 2). Here, it is worthy to differentiate between variable features [21, 22] that are those that require to provide a value (e.g., integer, string, float) during configuration; and *features with attributes* [10], which can be used to model NFPs such as the performance or price of features (see Execution time and Energy consumption attributes of the RSA feature in Figure 2). The incorporation of clonable and variable features brings into play the definition of complex cross-tree constraints. Examples are modeling the dependency between features that are children of different clonable features, or a dependency involving a numerical value (see cross-tree constraints in Figure 2). Thus, a requirement about variability modeling is:

**DA.Req1.** *Support for complex variability modeling including clonable features, variable features, features with attributes, and complex constraints.*

WeaFQAs also considers NFPs such as performance and energy efficiency in the context of an FQA configuration. Although dealing with NFPs in feature models as attributes [10] is a wide accepted approach [27], it presents some issues. Firstly, NFPs usually compete and conflict with each other. For example, using the AES encryption algorithm has a high energy efficiency but provides a poor

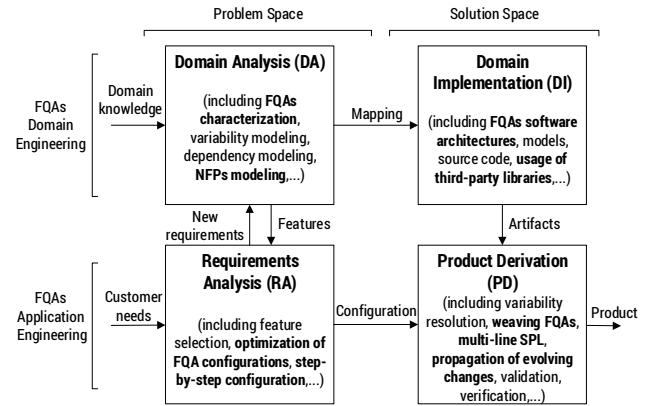


Figure 1: The WeaFQAs SPL process.

performance in execution time [34]. These relationships cannot be handled with the basic constraints of feature models (e.g., includes and/or excludes) and require more specific treatments (e.g., use of the NFR+ Framework) [18]. Secondly, sometimes the management of NFPs needs to be postponed until the requirements analysis or even product derivation phase because the NFPs' information about a feature is not available until the product is completely generated and tested, unless domain experts provide predictions and estimations of them. For example, to obtain the real energy efficiency or memory footprint of a product is necessary to evaluate it as a whole configuration and not as an independent features.

**DA.Req2.** *Support for dealing with and managing NFPs that can be associated to individual features or complete configurations.*

These two requirements appear in other current domains, such as the IoT [35, 55], where clonables features are essential to model the variability of the different devices, and NFPs need to be modeled when deployments have to be generated according to the tradeoff between different NFPs, such as latency and battery consumption.

Another challenge that WeaFQAs poses is the size of its variability model. Considering 20 FQAs modeled with a total of 178 features and 23 constraints, there are  $5.72e24$  configurations. Variability models of this size are unmanageable and thus is desirable to modularize them, as for example, using *composite variability models* as defined for the CVL language [22] (see Context feature in Figure 2). Another solution to modularize an SPL that could fit very well with WeaFQAs is the use of a *multi product line* (MultiPLs) approach [58]. In a MultiPL approach, the FQAs for each quality attribute (e.g., the encryption, authentication and hashing FQAs for the security attribute) would be defined in separated SPLs. These SPLs would then be composed when used in a specific application.

**DA.Req3.** *Support for modeling/managing huge feature models.*

**Requirements Analysis (RA).** WeaFQAs allows creating the FQAs configurations according to the application requirements, but also generating optimum configurations based on NFPs, such as performance or energy efficiency. Generating optimum configurations is an intractable problem when dealing with huge variability models or models containing variable features. As discussed in

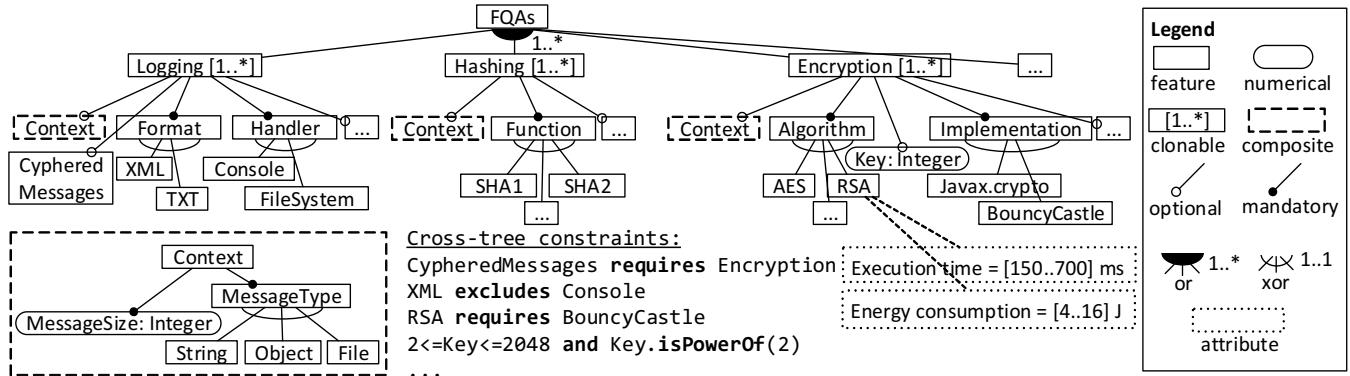


Figure 2: Excerpt of the WeaFQAs' variability model (complete model available in [36]).

DA. Req2, sometimes it is necessary to generate several configurations (or even all) in order to evaluate them and finding the best configuration before delivering the final product, or to reason about the system's variability. In those cases, specific formalizations of variability models (e.g., CNF [10] or BDD [25]) and reasoners and solvers (e.g., SAT [67], CPS [54]) need to be used depending on the type of analysis (e.g., calculate number of configurations, finding the optimum configuration, or generate all configurations).

**RA.Req1.** *Support for analyzing and generating optimum configurations from huge space configurations, based on different criteria (e.g., NFPs).*

In this phase, another activity that WeaFQAs considers is assisting application engineers when choosing a configuration. In this sense, WeaFQAs provides advises about the most appropriate selections based on the application engineers' goals. For example, to decide the most secure encryption algorithm or the most efficient framework that implements the RSA algorithm. This kind of assistant requires to manage partial configurations that will be completed in a step-by-step process.

**RA.Req2.** *Support for partial and step-by-step configurations.*

**Domain Implementation (DI).** There are frameworks and third party libraries that provide implementations of FQAs ready to be used, such as the Java Security package, the Apache Commons library, and the Spring Framework. So, implementing the FQAs' artifacts from scratch using a specific variability mechanism such as FOP or AOP is not an advisable option. Instead, the challenge posed by WeaFQAs is to handle the variability of existing artifacts in order to configure their functionality according to the selected features during the RA phase. Furthermore, the recurrent nature of the FQAs makes them suitable to be used in many different domains. Domains like web engineering involve multiple types of programming languages (e.g., JavaScript, Python, Groovy), and markup languages (e.g., HTML, CSS, XML) where applying typical variability development paradigms of SPL such as FOP or AOP is extremely difficult or even impossible [30].

**DI.Req1.** *Support for using and combining different variability mechanisms (FOP, AOP, annotations,...) independently from the language, and applying them to existing implementations.*

Additionally, WeaFQAs supports the management of FQAs at the architectural level. This means that WeaFQAs can handle architectural configurations of FQAs defined in any modeling language based on MOF (Meta-Object Facility), by using Model-Driven Engineering.

**DI.Req2.** *Support for managing artifacts variability at different abstraction levels: from software architecture models to code.*

**Product Derivation (PD).** In this phase, beyond the automatic generation of the product (here FQAs configurations), WeaFQAs has a specific need: the *weaving* of the FQAs into the final application. Following the WeaFQAs approach, where FQAs are modeled separately from the application, each generated configuration needs to be incorporated (woven) in different places of the application. In WeaFQAs this weaving activity can be performed together with the product derivation or in an independent activity (e.g., following a MultiPL [58]). In both cases, WeaFQAs supports the weaving process at the architectural level (using MOF-compliant models) or at the code level (by defining custom transformations) [29].

**PD.Req1.** *Support for weaving products or multi product lines.*

Finally, an important activity in an SPL process is the evolution engineering. When the applications' requirements change and/or the technology of the FQAs evolves, domain knowledge and artifacts need to be updated and the changes need to be propagated to the different deployed configurations. In some domains, as for example multi-tenants applications [32], with hundreds of configurations deployed, this activity should be performed automatically.

**PD.Req2.** *Support for automatic propagation of evolving changes to the deployed products.*

In the rest of the paper we take into account these requirements in order to answer our research questions.

## 4 STATE OF PRACTICE

This section answers our first research question:

**RQ1:** *What are the tools that provide some kind of support for SPLs? Are they available, usable, and keep up to date?*

We analyze the current state of practice of SPL tools to identify which ones are available online, and are really usable for industry,

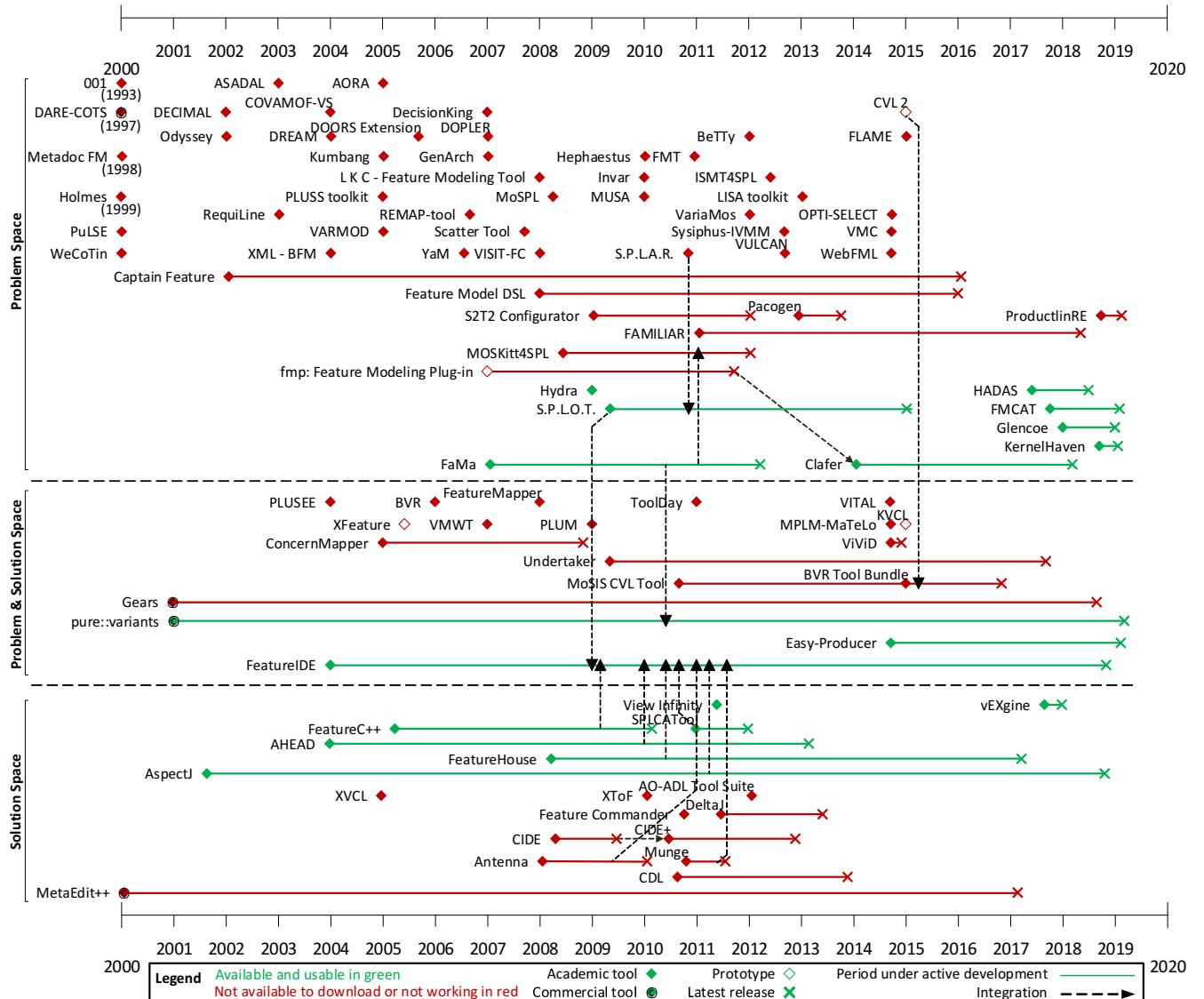


Figure 3: State of practice of SPL tools.

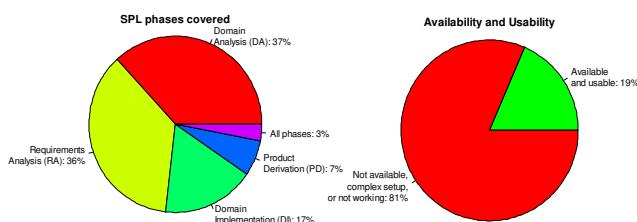


Figure 4: Summary of existing tools.

practitioners, and the SPL community. The goal is to collect all possible tools related with SPL to check their status before considering

them to analysis. This does not pretend to be a systematic review of tools, but an in-depth study to identify existing tools.

**Research method.** We performed a manual search on different sources. First, we identified SLRs [6, 51] and surveys [43] about SPL tools. We also searched the proceedings of the demonstration and tool track in some of the most relevant events about SPL and variability (e.g., SPLC, VaMoS), for the period not covered by the SLRs and surveys (2015–2018). Second, for each reported tool we searched for its availability (i.e., its website, code repository or executable). When the information was not available in the paper we performed a manual search on web search engines (e.g., Google) to localize the tool by applying the following search strings: <<name of the tool>>, tool, SPL, Software Product Line, and variability.

Finally, we glanced at the tool by downloading, installing, and launching it to check its correct functioning.

**Data extraction form.** We used Google Forms<sup>2</sup> to capture the basic information about the availability of the tools: name, brief description, URL, main reference, SPL's phases covered, type of tool (academic, commercial, prototype), first and last release date, availability, current status and integration with other tools. This data has been extracted from the information found in reference papers, the official website, and the code repository of the tool. The only inclusion criteria for this form is that the tool is directly related with SPL or is being used in the context of SPL.

**Result.** To illustrate the state of practice, we have built a timeline (Figure 3) with all SPL tools published until February 2019<sup>3</sup>. We found 97 tools. As summarized in Figure 4, only 3% of them cover all phases of the SPL process. Moreover, there seem to be more interest in the problem space than in the solution space since the DA (37% of the tools) and the RA (36%) are the phases most covered by the tools. The DI and PD phases are only covered by 17% and 7% of the tools respectively. The main problem with SPL tools is the fact that only 19% are available online to be downloaded and used, have an easy installation process or can be directly used online in a web browser (green timelines in Figure 3). Characteristics that make them attractive to be used in the community and industry. The other 81% are obsolete tools, have complex installation process, work with errors, or they are not available at all (red timelines in Figure 3). This evidences that there are lot of tools but most of them are academic or prototypes tools that are abandoned shortly when the associated research project ends.

The state of practice gives a wide vision of the current state of art of the SPL tools and helps practitioners to select appropriate tools. The main artifacts developed that allows replicating and/or improving this state of practice are available in [36].

## 5 TOOL SUPPORT FOR COMPLEX SPLS

This section answers our second research question, selecting first a subset of the tools identified in Section 4:

**RQ2:** *How the tools behave when they are used in complex SPLs?*

### 5.1 Tool selection

Four inclusion criteria (IC) are considered relevant to answer RQ2:

- IC1:** The tool is available online to be downloaded or used.
- IC2:** The installation process is straightforward and does not require complicate settings, additional dependencies or third-party plugins that can be obsolete.
- IC3:** The tool is under active development or the latest release is a final stable version.
- IC4:** The tool covers at least one of the main phases of the SPL process defined in Section 3 and in [4, 52].

Five exclusion criteria (EC) were used to exclude tools that we do not consider appropriate to be used in a professional environment:

- EC1:** The tool is a prototype or a preliminary or beta version.
- EC2:** The tool does not work or works with errors.

<sup>2</sup><https://forms.gle/JfH9bKHHTgCLc31R7>

<sup>3</sup>A .csv file with the tools information is available in [36].

**EC3:** The tool is commercial and the owner does not provide an alternative free limited version.

**EC4:** The tool is a compilation of independent tools not related to SPL (e.g., a CASE tool).

**EC5:** There is another similar, more actual or useful tool, or the tool has been integrated within another.

By applying our inclusion and exclusion criteria we have chosen the SPL tools to be analyzed in this section (Table 1). Note that there are many other tools that are available and usable (e.g., FeatureHouse [5], AHEAD [9,...]), but they have been excluded by EC5 since they are integrated within other tools like FeatureIDE [64]. Others are very updated (e.g., ProductlineRE [26]) but have so many obstacles to be installed, so they do not pass IC2. Others are exclusive for a specific domain (e.g., FMCAT [7] focusing on the analysis of dynamic services product line). Others are generic tools that are not specific of SPL, even if they are used as part of some SPL tools in concrete phases of the SPL engineering process (e.g., AspectJ to implement artifacts following an AOP approach).

### 5.2 Tool analysis

In this section we analyze the selected tools to check whether or not they satisfy the requirements of a complex SPL process like the WeaFQAs process presented in Section 3. All artifacts developed and used throughout the different phases to test the tools are available to repeat the experiments in [36]. This includes the variability models in different formats and the implementation code of the artifacts.

**5.2.1 Domain Analysis (DA) phase.** Apart from variability and dependency modeling, the DA process should take into account requirements DA.Req1, DA.Req2, and DA.Req3.

**Experiments.** We have modeled the variability of the FQAs (Figure 2) with the selected tools. The WeaFQAs variability model includes clonable features (Encryption [1..\*]), variable features (MessageSize: Integer), features with attributes (Execution time and Energy consumption of the encryption algorithms), composite units to modularize the variability model and to facilitate its management and configuration (Context), arbitrary group multiplicity (1..\*), and complex cross-tree constraints like those involving variable features.

**Analysis.** Regarding DA.Req1., all tools support basic feature models (mandatory and optional features, alternative (xor) and or groups, requires and excludes constraints). However the support for advanced characteristic is very limited (Table 2). Clonable features is the most difficult characteristic to be implemented, and thus, no tool provides support for them completely. Clafer allows cloning any feature in the variability model and configuring each instance, but this is done at the configuration step and deciding whether a feature is clonable or not should be done at the domain analysis phase. FeatureIDE and pure::variants allow a similar behaviour of clonable features by inserting subtrees in the variability model. In FeatureIDE, this characteristic follows the VELVET approach of MultiPLs [58], but it is still a prototype that only supports the FeatureHouse [5] composition approach when generating code, it contains many errors and with not enough documentation. Pure::variants introduces the concept of “variant instance” as a link in the feature model to another configuration space. In contrast to

**Table 1: Description of the selected SPL tools.**

Tool	Year	Last update	SPL phases	Description
S.P.L.O.T. [45]	2009	Jan. 2015	DA, RA	Online tool to edit, debug, analyze, configure, share and download feature models. It offers hundreds of feature models from academics and practitioners. Available in: <a href="http://www.splot-research.org/">http://www.splot-research.org/</a>
Glencoe [2]	2018	Jan. 2019	DA	Web application to work with variability models. Model importation from DIMACS or pure::variants [61]. Several solvers for the automated analysis of FMs. Available in: <a href="https://glencoe.hochschule-trier.de/">https://glencoe.hochschule-trier.de/</a>
Clafer [3]	2014	Feb. 2018	DA, RA	General-purpose lightweight language for structural modeling: feature modeling and configuration, class and object modeling, and metamodeling. Several solvers and model reasoners. Available in: <a href="https://www.clafer.org/">https://www.clafer.org/</a>
FeatureIDE [42, 64]	2004	Nov. 2018	DA, RA, DI, PD	Open-source Eclipse framework with plug-in based extension mechanism to integrate and test existing tools and SPL approaches [39] (FeatureHouse, AHEAD, AspectJ, Antenna, etc.). Available in: <a href="http://www.featureide.com/">http://www.featureide.com/</a>
pure::variants [61]	2003	Dec. 2018	DA, RA, DI, PD	Commercial solution that supports all phases of the SPL process. Many extensions that connect pure::variants with common systems and software engineering tools [12]. Available in: <a href="https://www.pure-systems.com/">https://www.pure-systems.com/</a>
vEXgine [33]	2017	Jan. 2018	PD	Customizable implementation of the CVL execution engine [69] that can be extended with custom transformation engines to support multiple variability approaches. Available in: <a href="http://caosd.lcc.uma.es/vexgine/">http://caosd.lcc.uma.es/vexgine/</a>

Clafer, the number of instances for the clonable feature has to be decided in the domain analysis phase and not at the configuration step, where this decision is normally taken.

The support for variable features and for feature with attributes is confused because of the thin difference between them. Clafer allows defining variable features with a specific type (e.g., integer) that behaves as a normal feature but allows providing a value during the configuration step, and also specifying constraints about the value of that feature. However, to support attributes in Clafer (as for example to specify a utility value for each feature) we have to rely in the Clafer Multi-Objective Optimizer (ClaferMOO [3]) that is a specific reasoner for attributed-feature models, or modeling the attributes as variable (numerical) features. This implies to define an additional variable feature (e.g., integer) for each normal feature in the variability model, and make sure those variable feature are selected in the final configuration. Contrary, pure::variants offers complete support for attributes but not for variable features that in this case can be implemented as attributes. FeatureIDE supports attributes only partially, because it requires selecting the composer “Extended Feature Modeling” and then, no other composer can be selected. Neither S.P.L.O.T. nor Glencoe support clonable features, variable features, and feature with attributes.

Finally, each tool provides additional characteristics for variability modeling. For instance, Glencoe and pure::variants allow mixing mandatory features within “or” groups. Glencoe, Clafer and pure::variants support arbitrary multiplicity in group features (e.g.,  $x \dots y$  where  $x$  can be distinct from 1 and  $y$  distinct from \*). FeatureIDE and Clafer allow defining abstract features. Clafer (with constraints involving variable features) and pure::variants (with Prolog, and its own variant of OCL: pvSCL [61]) allow defining complex constraints.

Concerning DA.Req2, no tool provides explicit support for dealing with NFPs, relying on features with attributes to manage NFPs.

Respecting DA.Req3, first, huge feature models cannot be easily modularized within existing tools. Clafer allows defining multiple feature models as abstract classes but all of them in the same file. FeatureIDE, as discussed for clonable features, supports MultiPLs but it is in its infancy and the feature models itself cannot be divided in multiple files. In pure::variants, the support is better since it defines an Hierarchical Variant Composition to link a feature

**Table 2: Tool support for the Problem Space: DA and RA.**

	Requirements and Characteristics	S.P.L.O.T.	Glencoe	Clafer	FeatureIDE	pure::variants
<b>DA.Req1</b>						
Basic FMs (optional, mandatory, or, xor, requires, excludes)		■	■	■	■	■
Cardinality-based FMs (clonable features or features cloning/cardinalities)		□	□	□	□	□
Variable features (variable features with type – i.e., integer, string...)		□	□	■	□	□
Extended FMs (features with attributes)		□	□	□	□	■
Other extensions (complex constraints, arbitrary group multiplicity, abstract features...)		□	□	■	□	□
<b>DA.Req2</b>						
Support for NFPs		□	□	□	□	□
<b>DA.Req3</b>						
Modularization of FMs (composition units, multiple variability models)		□	□	□	□	■
Evolution of FMs (modification of features – e.g., change variability type, move feature...)		□	□	■	□	■
<b>RA.Req1</b>						
Analysis of FMs (statistics, validation...)		■	■	□	■	■
Number of configurations (model counting, independently of the FM's size)		■	■	□	□	■
Generation of configurations (enumerate all configurations)		□	□	■	■	□
Optimization of configurations (e.g., based on NFPs)		□	□	□	□	□
<b>RA.Req2</b>						
Partial configurations		■	□	■	□	□
Step-by-step configuration		■	□	□	□	□

■ Full support. □ Partial support. □ No support.

**Table 3: Tool support for the Solution Space: DI and PD.**

Requirements/Characteristic		FeatureIDE	pure::variants	vExgine
Domain Implementation (DI)	<b>DI.Req1</b>			
	Different variability mechanism (FOP, AOP, annotations,...)	■	■	□
Product Derivation (PD)	Multi-language / Language independent (used in the same project)	□	■	■
	<b>DI.Req2</b>			
Product Derivation (PD)	Model abstraction level (architecture, design, code,...)	□	□	□
	<b>PD.Req1</b>			
Product Derivation (PD)	Product derivation (variability resolution)	■	■	■
	Weaving products	□	□	■
	Multi Product Lines	□	□	□
Product Derivation (PD)	<b>PD.Req2</b>			
	Evolution changes (automatic propagation of changes)	□	□	□

■ Full support. □ Partial support. □ No support.

model inside another one. Second, modifications of the variability models once created can be complex in some tools like S.P.L.O.T. and Glencoe, where modifying a part of the feature model usually can be only achieved by removing that part and adding it again. Contrarily, Clafer and pure::variants allow even moving features from a branch to another in a straightforward way.

**Findings.** Table 2 summarizes the characteristics supported by the analyzed tools. S.P.L.O.T. and Glencoe are the most usable tools for the DA phase since they are available online, intuitive and easy to use and even their models can be exported to FeatureIDE and pure::variants, respectively. However, they do not provide any support for advanced characteristics. Only Glencoe and FeatureIDE use the notation proposed by Czarnecki [23] that is the most comprehensible and flexible by now (and the most used) [10]. The notation of Clafer can be tedious for variability modeling although it provides good support for variable features and acceptable support for clonable features. S.P.L.O.T. and pure::variants share a similar interface to build the feature model, following a tree structure but each of them with its own notation. It is worthy to mention that there are other tools that provide explicit support for clonable and variable features such as the tools that provide support to the CVL language [69] (e.g., MoSIS CVL Tool [63] and BVR Tool Bundle [65]). However, those tools do not meet our inclusion/exclusion criteria because they are not currently available or are not in a usable state.

**5.2.2 Requirements Analysis (RA) phase.** The goal of this process is to select a desired combination of FQAs according to the application requirements. This phase should also consider the optimization of the FQA configurations based on NFPs, as well as assisting application engineers when generating the configurations.

**Experiments.** By using the selected tools, we analyze the possible configurations of the FQA's variability model, generating different (or all possible) configurations, and finding optimum configurations based on the NFPs (e.g., performance and energy efficiency in WeaFQAs) when possible. The challenge in this step is dealing with huge variability models such as the FQAs feature model.

**Analysis.** Regarding RA.Req1., almost all tools provide some kind of support for analyzing the variability model. This means statistics and metrics about the variability model (core features, optional features, number of constraints,...), model validation (consistency, void feature model,...), and anomalies detection (dead features, false-optimal features, redundancy constraints,...). Clafer can only validate the model syntactically since it is a text plain modeling language with a formal grammar, but no further analysis about variability is carried out by default.

Depending on the requested analysis, each tool uses a specific feature model formalization and/or solver to perform the analysis. For example, to calculate the number of configurations or variability degree of the feature model, S.P.L.O.T. uses a Binary Decision Diagrams (BDD) engine [25] for which counting the number of valid configurations is straightforward. Glencoe uses a Sentential Decision diagram (SDD) [50] engine that enables to determine the total number of configurations within very short times. Within pure::variants is also possible to calculate the number of configurations for each subtree under a selected feature. However, these tools calculate the number of configurations without taking into account variable features, which considerably increments the total number of configurations. The other tools (Clafer and FeatureIDE) require to generate all configurations in order to enumerate them, and thus, with these tools is not possible to calculate the number of configurations for huge models, like the FQAs feature model, in a reasonable time. For instance, using the Choco solver [54] integrated in Clafer, it takes 1 hour to generate  $13e6$  configurations from a total of  $5.72e24$  (calculated with S.P.L.O.T.), requiring more than a billion of years to generate all configurations. FeatureIDE, in addition, generates the associated code, so it requires more time.

Additionally, none of the selected tools provide support for finding optimum configurations in feature models. Only Clafer, with its ClaferMOO module, provides a multi-objective optimization mode, but this implies to use another kind of model not related with the Clafer's variability model. Even so, with Clafer and FeatureIDE, we could generate all configurations (for small models), evaluate them based on the NFPs (e.g., performance and energy efficiency) and then finding the optimum configurations by using an specific optimization tool [34]. Other techniques such as random sampling applied to SPL [49] and formalizations of the feature models such as CNF or the use of advanced SAT solvers [14, 67] will allow reasoning about these aspects or help to solve these issues with huge models, but this is out of scope of this paper.

With respect to RA.Req2., only S.P.L.O.T. and Clafer provide a reasonable good support to manage partial configurations and step-by-step configuration. S.P.L.O.T. provides validation and statistics of the partial configuration, but also auto-completion of the configuration with less features or the configuration with more features. This is done through an online step-by-step configuration assistant. Clafer allows generating configurations from a partial one

thanks to its instantiation process based on constraint definition. FeatureIDE and pure::variants allow generating partial configurations and calculate the number of valid configurations from that partial configurations, but they do not incorporate a guided process like S.P.L.O.T. to assist the user.

**Findings.** No tool allows generating all configurations efficiently for huge variability models like the required in WeaFQAs. Tools are able to calculate that the complete FQAs' feature model has  $5.72e24$  different configurations, but without taking into account variable features (e.g., numerical values). In fact, nowadays, with the existing tool support for SPL it is not possible to generate optimum configurations of products based on some criteria like NFPs [34, 49].

**5.2.3 Domain Implementation (DI) phase.** This phase focuses on the implementation of the variable artifacts (e.g., models, code).

**Experiments.** We have implemented some of the FQAs using different variability mechanisms, concretely AOP, FOP, and annotations. We have reused third party libraries like the Java Security package for the Hashing FQA, the BouncyCastle library for Encryption, and the SLF4J API for Logging.

**Analysis.** Regarding DI.Req1., FeatureIDE is the tool that provides best support for different variability mechanisms. Concretely, it supports FOP using the FeatureHouse approach or AHEAD, AOP with AspectJ, and annotations with Antenna (Java comments), Colligens (C preprocessor) or Munge (Android), among others [44]. However, it is not possible to combine different approaches in different parts of the application (e.g., annotations and AHEAD) or to use different languages (e.g., Java and JavaScript). Actually, only the combination of FeatureHouse with Java and AspectJ is supported.

The pure::variants tool also provides good support for AOP (e.g., AspectJ and AspectC++), and annotations (e.g., for Java, JavaScript, C++) with its own variation points system, but not for FOP. The *family model* [61] of pure::variants allows describing the variable architecture/code and to connect it via appropriate rules to the feature model. Nevertheless, most of the advanced options of pure::variants are only available in the commercial version. In the case of vEXgine, it is possible to use and combine different variability mechanisms but the resolution of that variability needs to be delegated to an external engine [33].

Concerning DI.Req2., FeatureIDE and pure::variants offer very good support for implementing the variability at the code level as discussed in DI.Req1, while vEXgine needs specific extensions to work at code level [30]. At a high abstraction level (architecture and design), both pure::variants and vEXgine offer the best support. However, pure::variants requires the commercial version to manage high abstract models (e.g., UML), and vEXgine requires to define the appropriate model transformations despite it supports any MOF-compliant model [33]. FeatureIDE offers the possibility to combine FeatureHouse and UML, but actually, this integration is not completely operable.

**Findings.** With existing tools, it is very difficult to apply the variability mechanisms (e.g., AOP, FOP) to third party libraries like those that implement FQAs, and the solution is usually encapsulating the behaviour of those libraries in entities of the specific approach (e.g., aspects or features) or implementing the FQAs from

scratch using a specific variability mechanism. Moreover, no tool supports an effective variability mechanism to be applied over several languages (Java, Python, JavaScript) in the same project.

**5.2.4 Product Derivation (PD) phase.** The product derivation phase is in charge of generating the final product (configurations of FQAs) by resolving the variability of the artifacts (FQAs) according to the selection of features made in the RA phase. Then, those configurations of the FQAs needs to be incorporated into the final application by some combination mechanism (weaving, or MultiPL). Also, the propagation of changes in the final products when the requirements changes or the domain artifacts evolve needs to be considered in this phase.

**Experiments.** We have generated the final products by resolving the variability of the FQAs with different configurations based on the configuration models specified in the RA phase. When possible, we have incorporated those configurations to existing applications. For example, weaving the FQAs' configurations to an electronic voting (e-voting) application [31]. Then, we have evolved the FQAs' variability model and try to update the generated configurations.

**Analysis.** Variability resolution and product derivation is achieved by all analyzed tools. A limitation in FeatureIDE is that only one composer (e.g., FeatureHouse, annotations) can be selected for an SPL application and thus the combination of different approaches requires to build and integrate a custom composer within FeatureIDE.

Apart from resolving the variability, PD.Req1. cannot be completely satisfied. In fact, only vEXgine provides complete support for weaving FQAs by defining custom model transformations [29]. The flexibility of pure::variants allows integrating other tools like Git to partially support mixing variants [59]. FeatureIDE integrates the VELVET approach [58] for MultiPL, but this is a prototype and in this case the product derivation is not fully operable.

Regarding PD.Req2. the support for propagating changes in the variability model to the existing configurations exists but is limited. In pure::variants, the source code of the product variants can be evolved by using merge operations from Git [28, 59]. Also, with the help of specific model transformations and evolution algorithms [31, 32], vEXgine can evolve the deployed artifacts, but the effort of defining those transformations is considerable.

**Findings.** Apart from the basic activity in this phase existing tools have not paid attention to advanced characteristics (e.g., weaving, MultiPL, evolution). However, those characteristic could be incorporate in some tools thanks to their extension mechanisms such as the possibility to define new composers in FeatureIDE [39] or the custom engines and model transformations of vEXgine [33].

## 6 SPL TOOLS ROADMAP

This section answers our third research question:

**RQ3:** *Is it possible to carry out a complete SPL process with the existing tool support?*

To answer RQ3, based on the analysis in the previous section, we define some practical roadmaps to completely carry out an SPL process with the existing tool support (Figure 5). The answer for RQ3 is that existing tools do support the complete process of SPL, but with many limitations. As shown in Figure 5, Roadmap 1 with

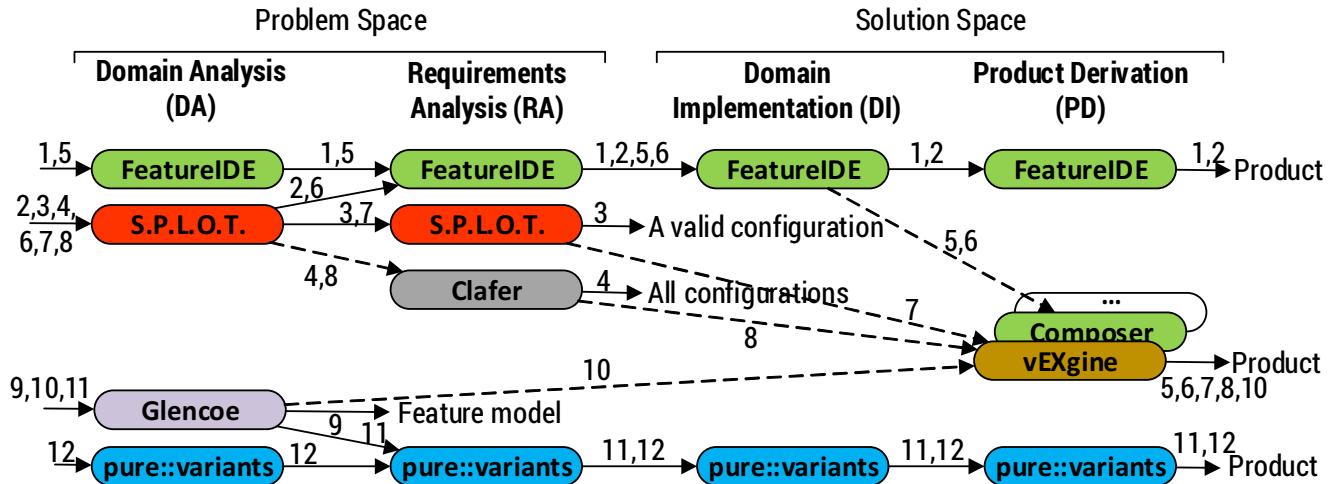


Figure 5: Roadmaps with the selected tools for an SPL process.

FeatureIDE and Roadmap 12 with pure::variants allows carrying out a complete SPL approach, covering the basic activities of an SPL process (variability modeling and artifacts implementation), and generating a final product. However, the limitations of these tools, as evidenced in Section 5, make them not suitable for complex SPL approaches like the WeaFQAs process that demands advanced SPL characteristics such as clonable features, managing huge models, or dealing with NFPs.

To partly solve these issues, SPL practitioners can combine some of the tools or integrate them. Following with our roadmaps (Figure 5), existing combinations are represented as solid lines, while possible combinations are represented as dashed lines. For instance, we can combine S.P.L.O.T. with FeatureIDE (Roadmaps 2 and 6) or Glencoe with pure::variants (Roadmap 11) to get the benefits of specifying the variability model in an online and easy to use web application like the offered by S.P.L.O.T. or Glencoe, and then loading the model in FeatureIDE or pure::variants, respectively. Since no standardized modeling format has been accepted after almost 30 years working with feature models, and the proposals for standardization (e.g., CVL [22], EMF [62]) have not jelled, each tool has defined its own format and notation, resulting in a high diversity of formats (e.g., SXFM, GUIDSL, Velvet, DIMACS,...). So, the roadmaps defined in this section will allow engineers and SPL practitioners to be aware about which tools can be used independently and in combination when a single tool does not support the complete SPL process.

In addition, when we are only interesting in analyzing the SPL variability, we can opt to use only Glencoe (Roadmap 9) that is the tool with best support for modeling and analyzing variability. When we need to generate an specific configuration (or a partial one) based on the requirements of the application, S.P.L.O.T. (Roadmap 3) offers an excellent feature-based interactive configuration module. When all configurations need to be generated at the RA phase, we can use Clafer (Roadmap 4). Note that we do not include a specific roadmap for Clafer because modeling the variability model in Clafer is a hard and tedious task that requires to learn a complex notation.

Instead, to cover Roadmaps 4 and 8 we have developed a feature model converter from S.P.L.O.T. to Clafer. The implemented scripts and algorithms to fill some of the possible connections between the roadmaps are available in [36].

For implementing the variable artifacts from scratch (i.e., following a proactive and/or a reactive approach to develop an SPL [19]), FeatureIDE is the recommendable choice because it allows using several variability approaches (FOP, AOP, annotations) despite the fact that it does not allow directly combining those approaches (except for AOP the combination of which is straightforward). For those domains in which the applications require to combine more than one different approach (e.g., web engineering), practitioners will need to implement specific composers to allow the combination work, like a new composer plugin for FeatureIDE (Roadmaps 5 and 6). In this sense, vEXgine (Roadmaps 7, 8 and 10) provides great flexibility because it is design to be extensible by means of model transformations. For an extractive approach where practitioners start with a collection of existing products [19], pure::variants is a good choice thanks to its family model that connects the existing artifacts with the feature model.

Regarding NFPs, although there are specific tools to deal with NFPs such as the NFR+ Framework [18], these tools are not intended to be used in an SPL, and they have to be integrated within other SPL tools. The same occurs with those approaches that manage NFPs in an ad-hoc way by associating features and/or configurations to NFPs stored in a database [34, 46]. Actually, no tool provides even good support for modeling attributes in the feature model and manage them through the SPL process, as for example to generate optimum configurations based on these attributes.

Finally, to deal with variability models at the architectural level, pure::variants is the most mature tool, with the only drawback that the commercial version of the tool is required [12]. Also vEXgine provides excellent support for resolving the variability of architectural models, but in this case the downside is that practitioners need to have some expertise in Model-Driven Engineering.

## 7 THREATS TO VALIDITY

This section discusses the threats to validity of this study [66]:

**Internal validity.** An internal validity concern is the reliability of the experiments to check the *functionality fulfillment* of tools. The functionality and characteristics analyzed vary among the tools. For example, clonable features are implemented differently in each tool. Literature reviews about tools usually study the support of functionalities as a primary goal. However, the goal of this paper is verifying how the tools satisfy the requirements in which we are interesting to carry out a complex SPL process, instead of reviewing all available functionalities provided by the tools.

**External validity.** An external validity concerns the *generalization/applicability of the results* to others SPL processes, beyond WeaFQAs. WeaFQAs was chosen because it follows the classical framework for SPL and incorporates additional requirements that can be found in current complex projects. Results of this work encompass from simpler SPLs [4, 52] to complex SPL processes like WeaFQAs or the Concern-Oriented Reuse (CORE) approach [1].

**Construct validity.** Construct validity relates to the completeness of our study, as well as any potential bias.

**Missing important tools in the state of practice.** The search for the tools information was conducted in several SLRs, proceedings of the most relevant conferences in SPL (e.g., SPLC) and variability (e.g., VaMoS), and in web search engines, and it was gathered through a data extraction form. We believe that we do not have omitted any relevant tool. However, since new tools are constantly appearing and evolving, we encourage practitioners to fill the information about any missed or new SPL tool in our form so that we can include them and continuously extend our study.

**Tools selection for analysis.** The defined inclusion and exclusion criteria to select the tools for our analysis can exclude some relevant tools (e.g., Gears). Our criteria focuses specially on the availability and usability of the tools that we consider the first obstacle for their adoption in small/medium organizations. Therefore, we omit those tools that are not available to be directly downloaded, require to pay a licence, or with inadequate documentation because those tools are not capable of being analyzed before acquiring them.

**Biased judgment selection and analysis.** Due to the researchers involved in this study are active in the SPL research area and produced related tools (e.g., vEXgine, HADAS, Hydra, AO-ADL), a validity problem could be author bias. Only vEXgine passed our inclusion/exclusion criteria. In addition, the decision to include vEXgine over other similar tools is threefold: (1) actually, it is the only available tool to provide support for CVL models [33]; (2) it is one of the few tools that work at the architectural level; and (3) it is very flexible to be extended or integrated within any other tool or approach. Despite those benefits, vEXgine also presents some limitations as discussed in Section 5.

**Conclusion validity.** Conclusion validity relates to the *reliability and robustness of our results*. A potential threat to conclusion validity is the interpretation of the results extracted from the analyzed tools. It was not always obvious to state from the empirical experiments if the tools completely or partially satisfy the exposed

requirements. To ensure the validity of our results, apart from the empirical experiments, we analyzed multiple data sources (e.g., tool documentation, reference papers, technical reports,...). Moreover, experiments were carried out at least by two primary authors that acted as reviewers of the results reported by the other.

## 8 CONCLUSIONS AND FUTURE WORK

We have presented a state of practice of the tools for SPL, focusing on their availability and usability. Based on this study we have later empirically analyzed the most usable tools in order to check out the existence of enough mature tool support for carrying out a complete SPL process with demanding requirements. We have defined up to 12 different roadmaps of the recommended tools to partially or completely support SPL activities, from the variability modeling until the product derivation phase.

The conclusion is that we need an integrated approach with appropriate tool support that covers all the activities/phases that are normally performed in complex SPLs. The main characteristics that tools should support are: (1) modeling variability of complex features (e.g., clonable features, variable features, composite features); (2) flexibility on the analysis of huge feature models considering optimization of configurations (e.g., analysis of NFPs); and (3) combination of multiple variability approaches (FOP, AOP, annotations), since only a variability approach (e.g., FOP) is not enough for some domains like web engineering that could greatly benefit from the use of SPLs. Therefore, with the existing tool support is possible to carry out a simple SPL process but tools present several limitations when dealing with complex SPLs.

As future work, we plan to continue our study to incorporate updated or new tools that could appear and that can be incorporated to our roadmaps.

## ACKNOWLEDGMENTS

This work is supported by the projects Magic P12-TIC1814, HADAS TIN2015-64841-R (co-financed by FEDER funds), MEDEA RTI2018-099213-B-I00 (co-financed by FEDER funds), and TASOVA MCIU-AEI TIN2017-90644-REDT.

## REFERENCES

- [1] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. 2017. Modelling a family of systems for crisis management with concern-oriented reuse. *Softw. Pract. Exper.* 47, 7 (2017), 985–999. <https://doi.org/10.1002/spe.2463>
- [2] Georg Rock Anna Schmitt, Christian Bettinger. 2018. Glencoe – A Tool for Specification, Visualization and Formal Analysis of Product Lines. In *Transdisciplinary Engineering Methods for Social Innovation of Industry 4.0 (Advances in Transdisciplinary Engineering)*, Vol. 7. 665–673. <https://doi.org/10.3233/978-1-61499-898-3-665>
- [3] Michał Antkiewicz, Kacper Bak, Aleksandr Murashkin, Rafael Olaechea, Jia Hui (Jimmy) Liang, and Krzysztof Czarnecki. 2013. Clafer Tools for Product Line Engineering. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops (SPLC'13 Workshops)*. ACM, New York, NY, USA, 130–135. <https://doi.org/10.1145/2499777.2499779>
- [4] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer. <https://doi.org/10.1007/978-3-642-37521-7>
- [5] S. Apel, C. Kästner, and C. Lengauer. 2013. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering* 39, 1 (Jan 2013), 63–79. <https://doi.org/10.1109/TSE.2011.120>
- [6] Rabih Bashroush, Muhammad Garba, Rick Rabiser, Iris Groher, and Goetz Botterweck. 2017. CASE Tool Support for Variability Management in Software Product Lines. *ACM Comput. Surv.* 50, 1, Article 14 (March 2017), 45 pages. <https://doi.org/10.1145/3034827>

- [7] Davide Basile, Felicita Di Giandomenico, and Stefania Gnesi. 2017. FMCAT: Supporting Dynamic Service-based Product Lines. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B (SPLC '17)*. ACM, New York, NY, USA, 3–8. <https://doi.org/10.1145/3109729.3109760>
- [8] Jonas Ferreira Bastos, Paulo Anselmo da Mota Silveira Neto, Pádraig O'Leary, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. 2017. Software product lines adoption in small organizations. *Journal of Systems and Software* 131 (2017), 112 – 128. <https://doi.org/10.1016/j.jss.2017.05.052>
- [9] D. Batory, J. N. Sarvela, and A. Rauschmayer. 2004. Scaling step-wise refinement. *IEEE Transactions on Software Engineering* 30, 6 (June 2004), 355–371. <https://doi.org/10.1109/TSE.2004.23>
- [10] David Benavides, Pablo Trinidad, and Antonio Ruiz Cortés. 2005. Automated Reasoning on Feature Models. In *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13–17, 2005, Proceedings*. 491–503. [https://doi.org/10.1007/11431855\\_34](https://doi.org/10.1007/11431855_34)
- [11] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '13)*. ACM, New York, NY, USA, Article 7, 8 pages. <https://doi.org/10.1145/2430502.2430513>
- [12] Danilo Beuche. 2016. Using Pure: Variants Across the Product Line Lifecycle. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC '16)*. ACM, New York, NY, USA, 333–336. <https://doi.org/10.1145/2934466.2962729>
- [13] Jan Bosch, Rafael Capilla, and Rich Hilliard. 2015. Trends in Systems and Software Variability. *IEEE Software* 32, 3 (2015), 44–51. <https://doi.org/10.1109/MS.2015.74>
- [14] Agustina Buccella, Matías Pol'la, Esteban Ruiz de Galarraga, and Alejandra Cechich. 2018. Combining Automatic Variability Analysis Tools: An SPL Approach for Building a Framework for Composition. In *Computational Science and Its Applications – ICCSA 2018*, Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Elena Stankova, Carmelo M. Torre, Ana María A.C. Rocha, David Taniar, Bernady O. Apduhan, Eufemia Tarantino, and Yonseung Ryu (Eds.). Springer International Publishing, Cham, 435–451.
- [15] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz Cortés, and Mike Hinckey. 2014. An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. *Journal of Systems and Software* 91 (2014), 3–23. <https://doi.org/10.1016/j.jss.2013.12.038>
- [16] Rafael Capilla, Alejandro Sánchez, and Juan C Dueñas. 2007. An analysis of variability modeling and management tools for product line development. In *Software and Service Variability Management Workshop-Concepts, Models, and Tools*. 32–47.
- [17] Lianping Chen and Muhammad Ali Babar. 2011. A systematic review of evaluation of variability management approaches in software product lines. *Information and Software Technology* 53, 4 (2011), 344 – 362. <https://doi.org/10.1016/j.infsof.2010.12.006> Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing.
- [18] Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. 2012. *Non-functional requirements in software engineering*. Vol. 5. Springer Science & Business Media.
- [19] Paul C. Clements and Charles W. Krueger. 2002. Point - Counterpoint: Being Proactive Pays Off - Eliminating the Adoption. *IEEE Software* 19, 4 (2002), 28–31. <https://doi.org/10.1109/MS.2002.1020283>
- [20] Katianna Constantino, Juliana Alves Pereira, Juliana Padilha, Priscilla Vasconcelos, and Eduardo Figueiredo. 2016. An Empirical Study of Two Software Product Line Tools. In *Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering (ENASE 2016)*. SCITEPRESS - Science and Technology Publications, Lda, Portugal, 164–171. <https://doi.org/10.5220/0005829801640171>
- [21] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2013. Beyond Boolean Product-line Model Checking: Dealing with Feature Attributes and Multi-features. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 472–481. <http://dl.acm.org/citation.cfm?id=2486788.2486851>
- [22] CVL Submission Team. 2012. Common variability language (CVL), OMG revised submission. <http://www.omgwiki.org/variability/lib/exe/fetch.php?id=start&cache=cache&media=cvl-revised-submission.pdf>.
- [23] Krzysztof Czarnecki. 2002. Generative Programming: Methods, Techniques, and Applications Tutorial Abstract. In *Software Reuse: Methods, Techniques, and Tools*, Cristina Gacek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 351–352.
- [24] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. 2005. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice* 10, 1 (2005), 7–29. <https://doi.org/10.1002/spip.213>
- [25] K. Czarnecki and A. Wasowski. 2007. Feature Diagrams and Logics: There and Back Again. In *11th International Software Product Line Conference (SPLC 2007)*. 23–34. <https://doi.org/10.1109/SPLINE.2007.24>
- [26] Javad Ghofrani and Anna Lena Fehlhaber. 2018. ProductlinRE: Online Management Tool for Requirements Engineering of Software Product Lines. In *Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 2 (SPLC '18)*. ACM, New York, NY, USA, 17–22. <https://doi.org/10.1145/3236405.3236407>
- [27] F. Z. Hammani. 2014. Survey of Non-Functional Requirements modeling and verification of Software Product Lines. In *2014 IEEE Eighth International Conference on Research Challenges in Information Science (RCIS)*. 1–6. <https://doi.org/10.1109/RCIS.2014.6861085>
- [28] Robert Hellebrand, Michael Schulze, and Uwe Ryssel. 2017. Reverse Engineering Challenges of the Feedback Scenario in Co-evolving Product Lines. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B (SPLC '17)*. ACM, New York, NY, USA, 53–56. <https://doi.org/10.1145/3109729.3109735>
- [29] Jose Miguel Horcas. 2018. *WeaFQAs: A Software Product Line Approach for Customizing and Weaving Efficient Functional Quality Attributes*. phdthesis. Universidad de Málaga. <https://hdl.handle.net/10630/17231>
- [30] Jose Miguel Horcas, Alejandro Cortiñas, Lidia Fuentes, and Miguel R. Luaces. 2018. Integrating the common variability language with multilanguage annotations for web engineering. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10–14, 2018*. 196–207. <https://doi.org/10.1145/3233027.3233049>
- [31] Jose Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2016. An automatic process for weaving functional quality attributes using a software product line approach. *Journal of Systems and Software* 112 (2016), 78–95. <https://doi.org/10.1016/j.jss.2015.11.005>
- [32] Jose Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2016. Product Line Architecture for Automatic Evolution of Multi-Tenant Applications. In *20th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2016, Vienna, Austria, September 5–9, 2016*. 1–10. <https://doi.org/10.1109/EDOC.2016.7579384>
- [33] Jose Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2017. Extending the Common Variability Language (CVL) Engine: A Practical Tool. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B (SPLC '17)*. ACM, New York, NY, USA, 32–37. <https://doi.org/10.1145/3109729.3109749>
- [34] Jose Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2018. Variability models for generating efficient configurations of functional quality attributes. *Information & Software Technology* 95 (2018), 147–164. <https://doi.org/10.1016/j.infsof.2017.10.018>
- [35] Jose Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2019. Context-aware energy-efficient applications for cyber-physical systems. *Ad Hoc Networks* 82 (2019), 15–30. <https://doi.org/10.1016/j.adhoc.2018.08.004>
- [36] Jose Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2019. WeaFQAs' resources and artifacts. <https://github.com/jmhhorcas/SPLC2019>
- [37] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßnich, Martin Becker, and Sven Apel. 2016. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering* 21, 2 (01 Apr 2016), 449–482. <https://doi.org/10.1007/s10664-015-9360-1>
- [38] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECCOP'97 – Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 220–242.
- [39] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. 2017. FeatureIDE: Empowering Third-Party Developers. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B (SPLC '17)*. ACM, New York, NY, USA, 42–45. <https://doi.org/10.1145/3109729.3109751>
- [40] Charles Krueger and Paul Clements. 2018. Feature-based Systems and Software Product Line Engineering with Gears from BigLever. In *Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 2 (SPLC '18)*. ACM, New York, NY, USA, 1–4. <https://doi.org/10.1145/3236405.3236409>
- [41] Miguel A. Laguna and Yania Crespo. 2013. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Science of Computer Programming* 78, 8 (2013), 1010 – 1034. <https://doi.org/10.1016/j.scico.2012.05.003> Special section on software evolution, adaptability, and maintenance & Special section on the Brazilian Symposium on Programming Languages.
- [42] Thomas Leich, Sven Apel, Laura Marnitz, and Gunter Saake. 2005. Tool Support for Feature-oriented Software Development: FeatureIDE: an Eclipse-based Approach. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange (eclipse '05)*. ACM, New York, NY, USA, 55–59. <https://doi.org/10.1145/1117696.1117708>
- [43] Liana Barachisio Lisboa, Vinicius Cardoso Garcia, Daniel Lucrélio, Eduardo Santana de Almeida, Silvio Romero de Lemos Meira, and Renata Pontin de Mattos Fortes. 2010. A systematic review of domain analysis tools. *Information and Software Technology* 52, 1 (2010), 1 – 13. <https://doi.org/10.1016/j.infsof.2009.05.001>
- [44] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer. <https://doi.org/10.1007/978-3-319-61443-4>

- [45] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 761–762. <https://doi.org/10.1145/1639950.1640002>
- [46] Daniel-Jesus Munoz, Mónica Pinto, and Lidia Fuentes. 2018. Finding correlations of features affecting energy consumption and performance of web servers using the HADAS eco-assistant. *Computing* 100, 11 (2018), 1155–1173. <https://doi.org/10.1007/s00607-018-0632-7>
- [47] N Nazar and TMJ Rakotomahefa. 2016. Analysis of a Small Company for Software Product Line Adoption—An Industrial Case Study. *International Journal of Computer Theory and Engineering* 4, 4 (2016), 313.
- [48] Lina Ochoa, Juliana Alves Pereira, Oscar González-Rojas, Harold Castro, and Gunter Saake. 2017. A Survey on Scalability and Performance Concerns in Extended Product Lines Configuration. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '17)*. ACM, New York, NY, USA, 5–12. <https://doi.org/10.1145/3023956.3023959>
- [49] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-optimal Configurations in Product Lines by Random Sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 61–71. <https://doi.org/10.1145/3106237.3106273>
- [50] Umut Oztok and Adnan Darwiche. 2015. A Top-down Compiler for Sentential Decision Diagrams. In *Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI'15)*. AAAI Press, 3141–3148. <http://dl.acm.org/citation.cfm?id=2832581.2832687>
- [51] Juliana Alves Pereira, Kattiana Constantino, and Eduardo Figueiredo. 2014. A Systematic Literature Review of Software Product Line Management Tools. In *Software Reuse for Dynamic Systems in the Cloud and Beyond*, Ina Schaefer and Ioannis Stamelos (Eds.). Springer International Publishing, Cham, 73–89.
- [52] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin, Heidelberg.
- [53] Christian Prehofer. 1997. Feature-oriented programming: A fresh look at objects. In *ECOOP 97 — Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 419–443.
- [54] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. 2017. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. <http://www.choco-solver.org>
- [55] Clément Quinton, Daniel Romero, and Laurence Duchien. 2016. SALOON: a platform for selecting and configuring cloud environments. *Software: Practice and Experience* 46, 1 (2016), 55–78. <https://doi.org/10.1002/spe.2311> arXiv:<https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2311>
- [56] Mikko Raatikainen, Juha Tiihonen, and Tomi Männistö. 2019. Software product lines and variability modeling: A tertiary study. *Journal of Systems and Software* 149 (2019), 485 – 510. <https://doi.org/10.1016/j.jss.2018.12.027>
- [57] Rick Rabiser, Klaus Schmid, Martin Becker, Goetz Botterweck, Matthias Galster, Iris Groher, and Danny Weyns. 2018. A study and comparison of industrial vs. academic software product line research published at SPLC. In *Proceedings of the 22nd International Conference on Systems and Software Product Line - SPLC '18*. 14–24. <https://doi.org/10.1145/3233027.3233028>
- [58] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. 2011. Multi-dimensional Variability Modeling. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '11)*. ACM, New York, NY, USA, 11–20. <https://doi.org/10.1145/1944892.1944894>
- [59] Sandro Schulze, Michael Schulze, Uwe Ryssel, and Christoph Seidl. 2016. Aligning Coevolving Artifacts Between Software Product Lines and Products. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '16)*. ACM, New York, NY, USA, 9–16. <https://doi.org/10.1145/2866614.2866616>
- [60] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. 2016. Extending Feature Models with Relative Cardinalities. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC '16)*. ACM, New York, NY, USA, 79–88. <https://doi.org/10.1145/2934466.2934475>
- [61] Olaf Spinczyk and Danilo Beuche. 2004. Modeling and Building Software Product Lines with Eclipse. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '04)*. ACM, New York, NY, USA, 18–19. <https://doi.org/10.1145/1028664.1028675>
- [62] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.
- [63] Andreas Svendsen, Xiaorui Zhang, Franck Fleurey, Øystein Haugen, Gørán K. Olsen, and Birger Møller-Pedersen. 2010. CVL Tool - Modeling Variability in SPLs. In *Software Product Lines - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13–17, 2010. Workshop Proceedings (Volume 2 : Workshops, Industrial Track, Doctoral Symposium, Demonstrations and Tools)*. 299.
- [64] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70 – 85. <https://doi.org/10.1016/j.scico.2012.06.002> Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).
- [65] Anatoly Vasilevskiy, Øystein Haugen, Franck Chauvel, Martin Fagereng Johansen, and Daisuke Shimbara. 2015. The BVR Tool Bundle to Support Product Line Engineering. In *Proceedings of the 19th International Conference on Software Product Line (SPLC '15)*. ACM, New York, NY, USA, 380–384. <https://doi.org/10.1145/2791060.2791094>
- [66] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. 2012. *Experimentation in Software Engineering*. Springer. <https://doi.org/10.1007/978-3-642-29044-2>
- [67] Yi Xiang, Yuren Zhou, Zibin Zheng, and Miqing Li. 2018. Configuring Software Product Lines by Combining Many-Objective Optimization and SAT Solvers. *ACM Trans. Softw. Eng. Methodol.* 26, 4, Article 14 (Feb. 2018), 46 pages. <https://doi.org/10.1145/3176644>
- [68] Guoheng Zhang, Huilin Ye, and Yuqing Lin. 2014. Quality attribute modeling and quality aware product configuration in software product lines. *Software Quality Journal* 22, 3 (2014), 365–401. <https://doi.org/10.1007/s11219-013-9197-z>
- [69] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. 2008. Adding Standardized Variability to Domain Specific Languages. In *2008 12th International Software Product Line Conference*. 139–148. <https://doi.org/10.1109/SPLC.2008.25>

# Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems

Daniel Strüber<sup>1</sup>, Mukelabai Mukelabai<sup>1</sup>, Jacob Krüger<sup>2</sup>, Stefan Fischer<sup>3</sup>,

Lukas Linsbauer<sup>3</sup>, Jabier Martinez<sup>4</sup>, Thorsten Berger<sup>1</sup>

<sup>1</sup>Chalmers | University of Gothenburg, Sweden, <sup>2</sup>University of Magdeburg, Germany, <sup>3</sup>JKU Linz, Austria, <sup>4</sup>Tecnalia, Spain

## ABSTRACT

The evolution of variant-rich systems is a challenging task. To support developers, the research community has proposed a range of different techniques over the last decades. However, many techniques have not been adopted in practice so far. To advance such techniques and to support their adoption, it is crucial to evaluate them against realistic baselines, ideally in the form of generally accessible benchmarks. To this end, we need to improve our empirical understanding of typical evolution scenarios for variant-rich systems and their relevance for benchmarking. In this paper, we establish eleven evolution scenarios in which benchmarks would be beneficial. Our scenarios cover typical lifecycles of variant-rich system, ranging from clone & own to adopting and evolving a configurable product-line platform. For each scenario, we formulate benchmarking requirements and assess its clarity and relevance via a survey with experts in variant-rich systems and software evolution. We also surveyed the existing benchmarking landscape, identifying synergies and gaps. We observed that most scenarios, despite being perceived as important by experts, are only partially or not at all supported by existing benchmarks—a call to arms for building community benchmarks upon our requirements. We hope that our work raises awareness for benchmarking as a means to advance techniques for evolving variant-rich systems, and that it will lead to a benchmarking initiative in our community.

## CCS CONCEPTS

- Software and its engineering → Software product lines; Software evolution;

## KEYWORDS

software evolution, software variability, product lines, benchmark

### ACM Reference Format:

Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19, September 9–13, 2019, Paris, France)*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3336294.3336302>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336302>

## 1 INTRODUCTION

Evolving a variant-rich software system is a challenging task. Based on feature additions, bugfixes, and customizations, a variant-rich system evolves in two dimensions: (1) in its variability when new variants are added over time, and (2) in each individual variant, as variants are continuously modified. From these dimensions, various evolution scenarios arise. For example, variability may be managed using clone & own [25], that is, by copying and modifying existing variants. In this case, changes performed on one variant are often propagated to other variants (*variant synchronization*). When the number of variants grows, a project initially managed using clone & own might be migrated to an integrated product-line platform [8, 13, 50], comprising a variability model [19, 38] and implementation assets with variability mechanisms (e.g., preprocessor annotations or composable modules). In this case, all assets in all variants that correspond to a given feature must be identified (*feature location*). Supporting developers during such scenarios requires adequate techniques, many of which have been proposed in recent years [2, 3, 7, 8, 10, 20, 27, 29, 37, 39, 48, 50, 60, 72, 77, 79, 87, 90, 94, 95].

The maturity of a research field depends on the availability of commonly accepted benchmarks for comparing new techniques to the state of the art. We define a benchmark as *a framework or realistic dataset that can be used to evaluate the techniques of a given domain*. Realistic means that *the dataset should have been initially created by industrial practitioners; it may be augmented with meta-data that can come from researchers*. In the case of evolving variant-rich systems, despite the progress on developing new techniques and tools, evaluation methodologies are usually determined ad hoc. To evaluate available techniques in a more systematic way, a common benchmark set has yet to emerge.

Inspired by a theory of benchmarks in software engineering [91], we believe that the community can substantially move forward by setting up a common set of benchmarks for evaluating techniques for evolving variant-rich systems. With this goal in mind, we follow typical recommendations for benchmark development [91]: to lead the effort with a small number of primary organizers, to build on established research results, and to incorporate community feedback to establish a consensus on the benchmark. As such, our long-term goal is to establish a publicly available benchmark set fulfilling the requirements of successful benchmarks [91]: clarity, relevance, accessibility, affordability, solvability, portability, and scalability.

In this paper, as a step towards this long-term goal, we lay the foundations for a benchmark set for evaluating techniques for evolving variant-rich systems. We conceive the scenarios that the benchmark set needs to support, show the relevance and clarity of our descriptions based on community feedback, and survey the state of the art of related datasets to identify potential benchmarks.

We make the following contributions:

- Eleven scenarios for benchmarking the techniques that support developers when evolving variant-rich systems (Sec. 2), including sub-scenarios, requirements, and evaluation metrics;
- A community survey with experts on software variability and evolution, focusing on the clarity and relevance of our scenarios (Sec. 3) and relying on an iterative, design-science approach;
- A survey of existing benchmarks for the scenarios (Sec. 4), selected upon our experience and the community survey;
- An online appendix with further information (e.g., links to benchmarks) and a replication package with the questionnaire and its data: <https://bitbucket.org/easelab/evobench>

We observed that various scenarios are only partially or not at all supported by existing benchmarks. We also identified synergies between scenarios and available benchmarks, based on the overlap of required benchmarking assets. Based on the positive feedback regarding the clarity and relevance of our benchmark descriptions, we believe that our work paves the way for a consolidated benchmark set for techniques used to evolve variant-rich systems.

## 2 EVOLUTION SCENARIOS

We establish eleven scenarios for techniques that support developers during the evolution of variant-rich systems. For each scenario, we argue how the relevant techniques can be evaluated with a benchmark. We introduce each scenario with a description, a list of more detailed sub-scenarios, a list of requirements for effective benchmarks, and a list of metrics for comparing the relevant techniques.

### 2.1 Methodology

To select the scenarios and construct the descriptions, we followed an iterative process involving all authors. We took inspiration from our experience as experts in software product line research, our various studies of evolution in practice [12, 13, 15, 17, 34, 35, 37, 42, 54, 56, 59, 67, 73, 74], and the mapping studies by Assunção et al. [8] and Laguna and Crespo [48]. Based on these sources, an initial list of scenarios emerged in a collaborative brainstorming session. Each scenario was assigned to a responsible author who developed an initial description. Based on mutual feedback, the authors refined the scenario descriptions and added, split, and merged scenarios and their descriptions. Each scenario description was revised by at least three authors. Eventually, a consensus on all scenario descriptions was reached. Afterwards, we performed a community survey to assess the clarity and relevance of the descriptions. The final version of the descriptions, as shown below, incorporates feedback from the survey (see the methodology description in Sec. 3).

### 2.2 Running Example

As a running example for the evolution of variant-rich systems, consider the following typical situation from practice.

Initially, a developer engineers, evolves, and maintains a single system, for instance, using a typical version-control system (e.g., Git). At some point, a customer requests a small adaptation. The developer reacts by adding a configuration option and variation points (e.g., based on if statements) in the code. Later, another customer requests a more complex adaption. The developer reacts

by copying the initial variant (i.e., creating a clone) of the system and adapting it to the new requirements (a.k.a., clone & own). Over time, further customers request specific adaptations and the developer uses either of these two strategies.

When the number of variants grows, this ad hoc reuse becomes inefficient. Namely, it becomes challenging and error-prone to identify which existing variant to clone and which parts (i.e., features) of other variants to incorporate in the new variant. The same applies to maintenance, as it is not clear which variants are affected by a bug or update. Any bug or update then needs to be fixed for each existing variant individually. Furthermore, an increasing number of configuration options challenges developers through intricate dependencies that need to be managed; and variation points clutter the source code, challenging program comprehension.

### 2.3 Scenario Descriptions

We now introduce our scenarios based on the running example, providing descriptions, sub-scenarios, benchmarking requirements and evaluation metrics. We focus on evaluation metrics that are custom to the scenario at hand. Some additional characteristics of interest, such as performance and usability, are important in *all* scenarios and should be supported by adequate metrics as well. Assessing the correctness or accuracy of a technique may require a *ground truth*, a curated, manually produced or (at least) checked set of assets assumed to be correct. Some scenarios involve the design choice of picking a metric from a broader class of metrics (e.g., similarity metrics); in these cases we specify only the class.

We visualize each scenario by means of a figure. Each figure provides a high-level overview of the respective scenario, representing the involved assets with boxes, techniques with rounded boxes, relationships with dashed arrows, and actions with solid arrows. In cases where a scenario has multiple sub-scenarios with varying kinds of assets, we show the superset of all required assets from all sub-scenarios. Each figure includes a large arrow on its left-hand side, indicating the direction of system evolution.

**Variant Synchronization (VS).** When evolving a variant-rich system based on clone & own, the developer frequently needs to *synchronize variants*. Bugfixes or feature implementations that are performed in one variant need to be propagated to other variants—a daunting task when performed manually. An automated technique (illustrated in Fig. 1) could facilitate this process by propagating changes or features contained in a variant [77, 78].

#### Sub-scenarios

- VS1: Propagation of changes across variants
- VS2: Propagation of features across variants

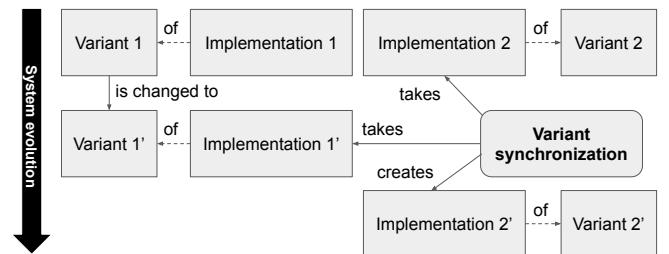


Figure 1: Variant synchronization (VS)

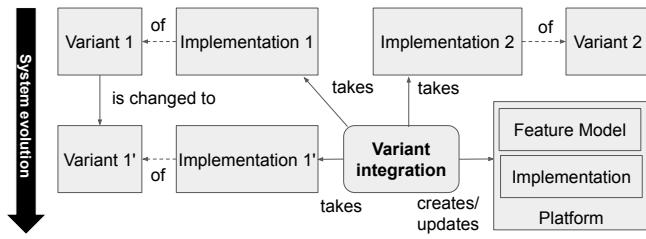


Figure 2: Variant integration (VI)

**Benchmark requirements**

- VS1/2: Implementation code of two or more variants
- VS1/2: Implementation code of variants after correct propagation (*ground truth*)
- VS1: Changes of at least one variant
- VS2: Feature locations of at least one variant

**Evaluation metrics**

- Accuracy: A metric for measuring the similarity between ground truth and computed variant implementation

**Variant Integration (VI).** Due to the drawbacks associated with clone & own [6, 25], a developer may deem it beneficial to manage the variant-rich system as a product-line platform. Such a platform comprises a variability model (e.g., feature [38] or decision model [19]) and implementation assets with a variability mechanism (e.g., preprocessor annotations or feature modules) that supports the on-demand generation of product variants. From the decision to move towards a product-line platform, two major *variant integration* tasks (a.k.a., extractive product-line adoption [43]) arise (illustrated in Fig. 2).

The first task is to enable the transition from the cloned variants to a platform [8]. Available techniques for this purpose take as input a set of products and produce as output a corresponding product-line platform [60]. Yet, further evolving the resulting platform can be challenging due to its variability—assets may be difficult to comprehend and modify. Therefore, the second task is to support extending and evolving a product line by means of individual, concrete product variants [51, 94]. This allows engineers to focus on concrete products during evolution to then feed the evolved product back into the platform to evolve it accordingly. Such techniques can be supported by variation control systems [51, 94] and approaches for incremental product-line adoption [6] from cloned variants.

**Sub-scenarios**

- VI1: Integrate a set of variants into the product-line platform
- VI2: Integrate changes to variants into the product-line platform

**Benchmark requirements**

- VI1: Set of individual variants
- VI2: Set of revisions of a product-line platform
- VI1/2: Product-line platform after correct integration (*ground truth*)

**Evaluation metrics**

- Accuracy: A metric for measuring the similarity between the ground truth and the computed product-line platform

**Feature Identification and Location (FIL).** Both, as an aid to better support clone & own development and to prepare the migration

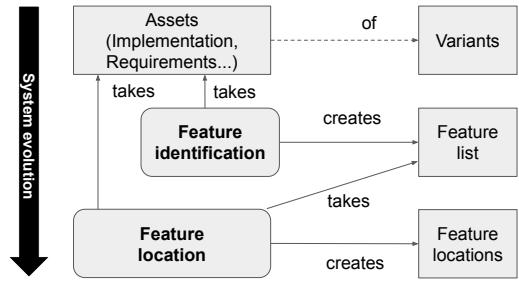


Figure 3: Feature identification and location (FIL)

to a product-line platform, developers may wish to determine which features exist in the system and which features are implemented in which assets (e.g., source code, models, requirements or other types of artifacts). For this purpose, they may rely on *feature identification* and *feature location* techniques (illustrated in Fig. 3). Feature identification aims to determine which features exist, whereas feature location aims to define the relationship of features to assets.

Feature identification is useful when the knowledge about features is only given implicitly in the assets, rather than explicitly as in a feature model. The objective is to analyze assets to extract candidate feature names. This can involve techniques to study domain knowledge or vocabulary of the considered domain, workshops to elicit features from experts [42], or automated techniques [61, 70, 100].

When done manually, feature location is a time-consuming and error-prone activity [45]. It has a long tradition for maintenance tasks (e.g., narrowing the scope for debugging code related to a feature), but is also highly relevant for identifying the boundaries of a feature at the implementation level to extract it as a reusable asset during re-engineering [47]. In this sense, it is related to traceability recovery. Feature location is usually expert-driven in industrial settings, however, several techniques based on static analysis, dynamic analysis, and information retrieval, or hybrid techniques, exist [8].

**Sub-scenarios**

- FIL1: Feature identification in single variants
- FIL2: Feature identification in multiple variants
- FIL3: Feature location in single systems
- FIL4: Feature location in multiple variants

**Benchmark requirements**

- FIL1/2/3/4: Assets representing variants, such as: implementation code, requirements, documentation, issue tracker data, change logs, version-control history
- FIL1/2/3/4: List of features (*ground truth* for FIL1/2)
- FIL3/4: Feature locations in sufficient granularity, such as files, folders, code blocks (*ground truth*)

**Evaluation metrics**

- Accuracy: Precision and Recall. Some authors in the literature use metrics, such as Mean Reciprocal Rank, that assess the accuracy of a *ranking* of results [18, 99].

**Constraints Extraction (CE).** In a variant-rich system, some features may be structurally or semantically related to other features. Initially, this information is not explicitly formalized, which makes it harder for the developer to understand these relationships. To this end, the developer may use an automated *constraints extraction* technique (illustrated in Fig. 4).

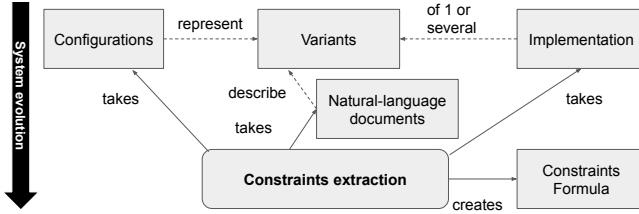


Figure 4: Constraints extraction (CE)

Constraints extraction is a core prerequisite for feature-model synthesis. However, even if the goal is not to obtain a model, explicitly knowing the constraints can help checking the validity of platform configurations, reducing the search space for combinatorial interaction testing (CIT, see below), and documenting features with their dependencies. The benchmark can be used to evaluate the extraction of constraints from various inputs, specifically, the product-line implementation (either code of individual variants or of a platform, [68, 69]), a set of example configurations [22], or natural-language artifacts, such as documentation. Over the development history, when a feature model exists, the constraints in the feature model would be annotated with their source (e.g., a def-use dependency between function definition and function call or domain dependency from hardware [69]). Considering cloned systems, constraints extraction can also be helpful to compare the variability that is implemented in different variants.

#### Sub-scenarios

- CE1: Constraints extraction from example configurations
- CE2: Constraints extraction from implementation code
- CE3: Constraints extraction from natural-language assets

#### Benchmark requirements

- CE1: Example configurations
- CE2: Implementation code of one or several variants
- CE3: Natural-language assets (e.g., documentation)
- CE1/2/3: Correct constraints formula (*ground truth*)

#### Evaluation metrics

- Accuracy: Similarity of configuration spaces (likely syntactic approximation; semantic comparison is a hard problem)

**Feature Model Synthesis (FMS).** To keep an overview understanding of features and their relationships, developers may want to create a feature model. *Feature model synthesis* (illustrated in Fig. 5) is an automated technique that can provide an initial feature model candidate. As input, it can rely on a given set of configurations, a set of variants (together with a list of features that each variable implements) or a product matrix to produce a feature model from which these assets can be derived.

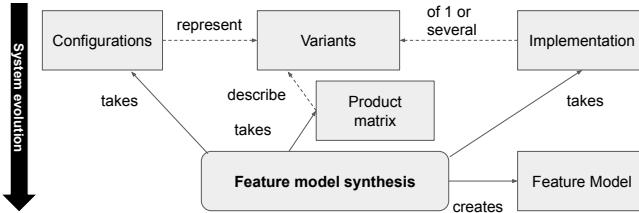


Figure 5: Feature model synthesis (FMS)

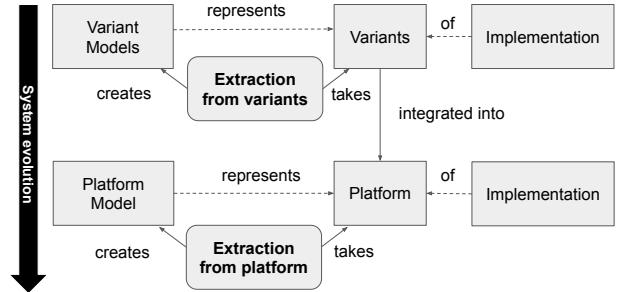


Figure 6: Architecture recovery (AR)

Various synthesis techniques [3, 86–88] are available. Their primary benefit is to identify a possible feature hierarchy, but they can also identify feature groups. Constraints extraction (CE, see above) can be incorporated as a component to identify constraints. *Sub-scenarios*

- FMS1: Feature model synthesis from a set of configurations
- FMS2: Feature model synthesis from an implementation
- FMS3: Feature model synthesis from a product matrix

#### Benchmark requirements

- FMS1: Example configurations
- FMS2: Implementation code of one or several variants
- FMS3: Product matrix
- FMS1/2/3: Correct feature model (*ground truth*)

#### Evaluation metrics

- Accuracy: Precision and Recall of recovered hierarchy edges and feature groups; similarity of the configuration spaces represented by the synthesized feature model and the input

**Architecture Recovery (AR).** When migrating cloned variants to a product-line platform, the developer may want to define a reference architecture for the resulting platform, using architectural models. Architectural models provide a different abstraction of the system structure than feature models, focusing on details and dependencies of implemented classes. *Architecture recovery* techniques (illustrated in Fig. 6) can extract architectural models automatically.

Various works [26, 41, 84, 92] focus on reverse engineering and comparing architectures from cloned variants to propose architectural models as a starting point for product-line adoption. Such models can include class, component, and collaboration diagrams that may be refined later on. For instance, the initial models may be used as input for a model-level variant integration technique, producing a platform model with explicit commonality and variability. Additional use cases include analyzing and comparing models to identify commonality and variability, or performing an automated analysis based on models.

#### Sub-scenarios

- AR1: Architecture extraction from a configurable platform
- AR2: Architecture extraction from a set of variants

#### Benchmark requirements

- AR1: Implementation code of one or several variants
- AR2: Implementation code of product line platform
- AR1/2: Correct architectural models (*ground truth*)

#### Evaluation metrics

- Accuracy: Similarity of extracted to ground truth models

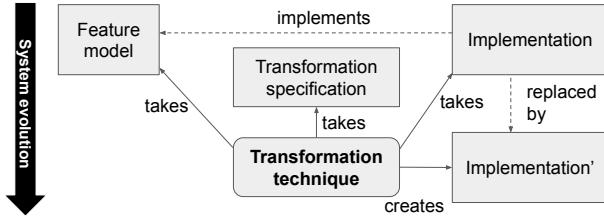


Figure 7: Transformations (TR)

**Transformations (TR).** To reduce manual effort during evolution tasks, such as refactoring or synchronization of multiple dependent assets in a variant-rich system, the developer may rely on *transformation* techniques. Transformation techniques are used to change system assets in an automated way. Tool support ranges from light-weight refactoring tools in IDEs to advanced model transformation languages with dedicated execution engines. Model transformations are used for manifold practical purposes, including translation, migration, and synchronization of assets [55].

When transforming a product-line platform (illustrated in Fig. 7), three sub-scenarios arise: First, to refactor the platform, improving its structure while behavior preservation is ensured for each variant [82]. Second, to partially refactor the platform [72] in such a way that only a controlled subset of all variants is changed. Third, to lift a given transformation from the single-product case to the platform, changing all variants consistently [80].

#### Sub-scenarios

- TR1: Refactoring of a product-line platform
- TR2: Partial refactoring of a product-line platform
- TR3: Lifting of a model transformation to a product-line platform

#### Benchmark requirements

- TR1/2: Product-line platform with feature model and implementation code
- TR3: Product-line platform with feature model and implementation model
- TR1/2/3: Transformation specification; for example, reference implementation
- TR1/2/3: Transformed implementation (*ground truth*)

#### Evaluation metrics

- Correctness: Number of errors
- Conciseness: Number of elements or lines of code of the given transformation

**Functional Testing (FT).** After evolving the variant-rich system, it is important to ensure it still behaves in the expected way. For instance, the variants that were available before the evolution should

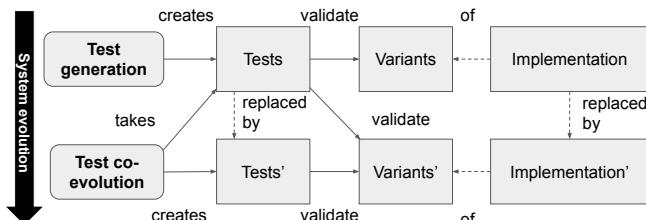


Figure 8: Functional testing (FT)

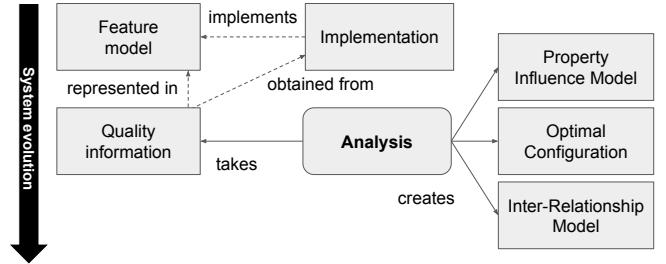


Figure 9: Analysis of non-functional properties (ANF)

still work after evolving the system. Regression testing aims to identify faults that may arise after the system has been changed and functionality does no longer work as before. *Functional testing* of variable software (illustrated in Fig. 8) adds challenges compared to conventional software testing, due to the variability that can influence the functionality of the variants.

For a product-line platform, we can divide testing into two phases: First, *domain testing* of common parts of the system. Second, *application testing* of variant-specific parts and interactions [24, 49]. In the case of clone & own, we can only do application testing for individual variants. To reduce testing effort, existing techniques aim to reuse test assets as much as possible. Assets from domain testing are reused in application testing, while trying to only test parts that are specific to selected variants to avoid redundancies. Similarly, it is useful to avoid redundancies after the evolution of the system, to test only parts relevant for the changes that have been applied. Moreover, for application testing it is unrealistic to test all possible variants. The most common technique used for the selection of variants is Combinatorial Interaction Testing (CIT), which identifies a subset of variants where interaction faults are most likely to occur, based on some coverage criteria [23]. Finally, evolution potentially makes some test cases outdated, because they no longer fit the evolved system. In such cases, system and tests must co-evolve [44].

#### Sub-scenarios

- FT1: Test generation for domain testing
- FT2: Test generation for application testing
- FT3: Test co-evolution

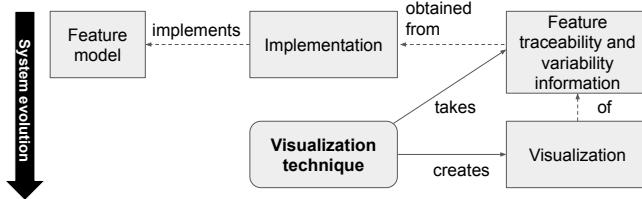
#### Benchmark requirements

- FT1/2/3: Implementation code from product line platform
- FT1/2/3: Known faults (*ground truth*)
- FT3: Tests to be co-evolved

#### Evaluation metrics

- Efficiency: Number of faults detected in relation to number of known faults
- Test effort: Number of tested variants, number of executed tests, execution time of tests only if all tests are executed on the same system, reuse of test assets

**Analysis of Non-Functional Properties (ANF).** Various non-functional or quality properties can be important for variant-rich systems, for example, performance in a safety-critical system [33], memory consumption in an embedded system with resource limitations [32], and usability aspects in human-computer interaction systems [58]. Therefore, the *analysis of non-functional properties*

**Figure 10: Visualization (VZ)**

in variant-rich systems (illustrated in Fig. 9) is crucial [67], as constraints on non-functional properties can be violated when the system evolves.

Developers would like to know the effect of specific features and feature interactions on the investigated quality property, particularly to identify possible improvements or regressions when changes were introduced. Such effects can be captured using a *property influence model* for the quality property under study, for instance, a performance influence model in the case of Siegmund et al. [89]. Also, an important analysis scenario is to identify *optimal configurations* that maximize one or multiple quality criteria while satisfying certain quality constraints [90]. This analysis is relevant for evolution when trying to balance various conflicting quality properties and understanding their relationships and trade-offs [76]. To this end, an *inter-relationship model* can be derived by analyzing the pareto front obtained during multi-criteria optimization. The considered analyses can be expensive, not only because of the combinatorial explosion in large systems, but also because computing non-functional properties can be a resource-intensive task.

#### Sub-scenarios

- ANF1: Analysis of impacts of features and feature interactions on quality properties
- ANF2: Optimization of configurations towards given quality criteria
- ANF3: Analysis of trade-offs between relationships among non-functional properties

#### Benchmark requirements

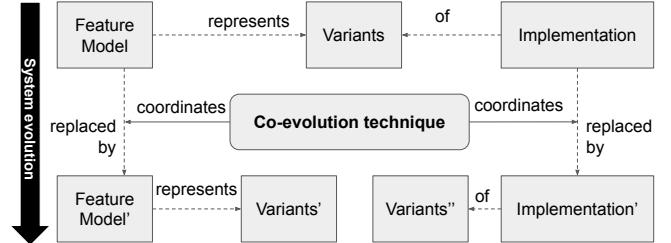
- ANF1/2/3: Feature model
- ANF1/2/3: Quality information, either given by annotations (e.g., extended feature models [11]), or by a method to calculate or estimate for a given product the quality metrics under study
- ANF1: Reference property influence model (*ground truth*)
- ANF2: Reference configuration (*ground truth*)
- ANF3: Reference inter-relationship model (*ground truth*)

#### Evaluation metrics

- Accuracy: Similarity between computed and reference model (ANF1/3), fitness of computed configuration in comparison to reference configuration (ANF2)

**Visualization (VZ).** To facilitate incremental migration [6] of clone & own-based variants to a product-line platform, the developer may want to visually inspect relations between features and implementation assets. Such a relation-visual inspection can be provided by *visualization techniques* (illustrated in Fig. 10).

During product-line engineering, visualizing variability in software assets can be useful for scenarios, such as product configuration [71, 76], testing (e.g., pairwise testing) [53], and constraint discovery [62]. Andam et al. [5] propose several feature-oriented

**Figure 11: Co-evolution of problem and solution space (CPS)**

views that exploit feature annotations [37] embedded by developers in the source code during development for tracing feature locations. A benchmark could be used to evaluate the effectiveness of several visualization techniques addressing the same sub-scenario. The main goal of benchmarking is to assess developer performance when using different techniques, which requires experimentation with human participants on selected development tasks.

#### Sub-scenarios

- VZ1: Visualizations for feature evolution and maintenance
- VZ2: Visualizations for constraint discovery
- VZ3: Visualizations for feature interaction assessment

#### Benchmark requirements

- VZ1/2/3: Implementation code with feature locations (preferably embedded feature traceability annotations, instead of only variability annotations for optional parts of source code)
- VZ1/2/3: Scenario-related tasks for developers, such as code comprehension and bug-finding tasks, based on generated visualizations

#### Evaluation metrics

- Developer performance: correctness, completion time in scenario-related tasks

**Co-Evolution of Problem Space and Solution Space (CPS).** After migrating the variant-rich system to a product-line platform and to further evolve it, the developer has to evolve both, the problem space (feature model) and the solution space (assets, such as architecture models and code). Evolving the solution space first can lead to outdated feature models that are inconsistent with the implementation. Evolving the problem space first limits the effects that changes to the implementation are allowed to have. To address these issues, an automated technique (illustrated in Fig. 11) may recommend *co-evolution steps* to keep both in sync.

For instance, when evolving the solution space first, the technique could extract updated feature dependencies (e.g., an additional dependency on another feature) based on their modified implementation (e.g., due to an additional method call) and suggest modifications to the problem space that reflect the changes made to the solution space. An important property is that problem space and solution space are consistent after every evolution step.

#### Sub-scenarios

- CPS1: Co-evolving the solution space based on problem space evolution
- CPS2: Co-evolving the problem space based on solution space evolution

#### Benchmark requirements

- CPS1/2: Product-line platform with feature model and implementation code
- CPS1/2: Sequence of revisions for feature model and implementation code (*ground truths*: implementation revisions for *CPS1*, feature model revisions for *CPS2*)

#### Evaluation metrics

- Accuracy: Similarity of computed and ground truth asset at a certain revision
- Correctness: Consistency between feature model and code

### 3 COMMUNITY SURVEY

To develop benchmarks, Sim et al. [91] suggest that incorporating community feedback is essential to establish consensus. We followed this recommendation by performing a questionnaire survey with members from the community on software variability and evolution. To gather feedback on the clarity and relevance of our scenario descriptions, two crucial quality criteria for a successful benchmark [91], our survey focused on two research questions:

**RQ<sub>1</sub>** How clear are our scenario descriptions?

**RQ<sub>2</sub>** How relevant are the described scenarios?

In the following, we report the details on our methodology, the results, and threats to validity.

#### 3.1 Methodology

We performed our questionnaire survey in March 2019. The participants for our survey were recruited from two sources: First, we contacted all participants (excluding ourselves) of a Dagstuhl seminar on variability and evolution, the two most relevant research areas (<https://dagstuhl.de/en/program/calendar/sempn/?semnr=19191>). Second, we contacted authors of recent papers on the same topic. We invited 71 individuals, 41 of them Dagstuhl participants. A total of 20 individuals completed our survey in the given timeframe.

Our questionnaire comprised three parts. First, we presented the general context of our benchmark, including the running example description we introduced in Sec. 2.2. Second, we described the eleven scenarios that we presented in Sec. 2. For each, we included the textual description as well as the list of sub-scenarios. We asked the participants to rate the clarity (using a 5-point Likert scale) of each scenario description (**RQ<sub>1</sub>**) with the question: *To which extent do you agree that the scenario is clearly described with respect to its usage context and purpose for benchmarking?* Then, we asked the participants to assess the relevance of each overall scenario and its sub-scenarios (**RQ<sub>2</sub>**) with the question: *To which extent do you agree that supporting the following sub-scenarios is important?* To assess the completeness of our descriptions, we asked the participants to name relevant sub-scenarios not yet considered. Finally, as a prerequisite for our survey of benchmarks (cf. Sec. 4), we asked the participants to name relevant benchmarks they were aware of. A replication package with the questionnaire and all data is found at: <https://bitbucket.org/easlab/evobench/>.

The initial responses to our survey pointed out a number of shortcomings in the scenario descriptions with respect to clarity. We used these responses to revise the questionnaire after the first 12 responses, presenting an improved version of the scenario descriptions to the remaining eight participants. This intervention is

justified by the methodological framework of design science [75], which emphasizes the continuous improvement of research artifacts based on actionable feedback, thus presenting a *best-effort approach*. The most significant change was to remove two sub-scenarios (one from the *variant synchronization* and one from the *transformation* scenario). In other cases, we reworded the scenario descriptions to add more explanations, examples, and avoid potentially confusing wording. To make sure that our revision indeed led to an improvement, we checked the clarity scores after the revision. We found that the clarity scores improved in all cases.

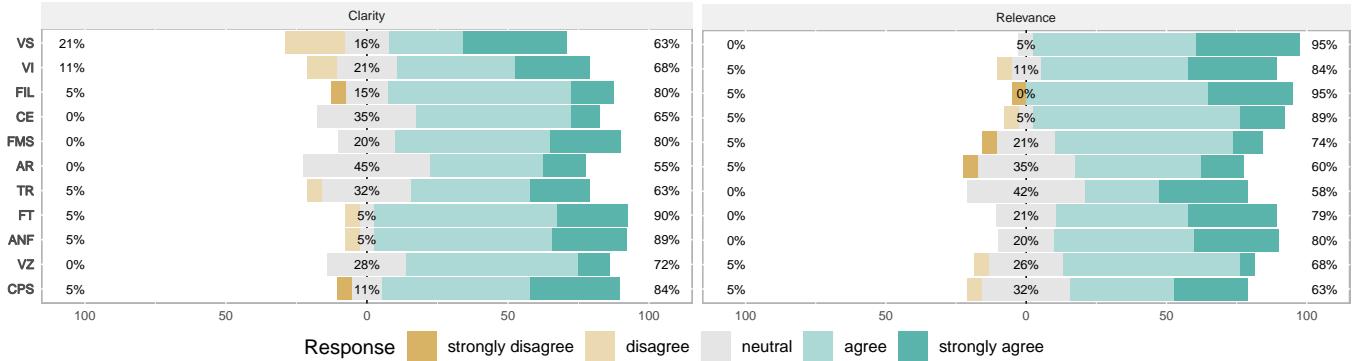
#### 3.2 Results

Figure 12 provides an overview of the results. For each scenario, we show the distribution of answers to our questions about clarity (**RQ<sub>1</sub>**) and relevance (**RQ<sub>2</sub>**). We further explain the results based on the textual feedback provided along with the answers.

**RQ<sub>1</sub>: Clarity.** For all scenarios, a majority of the participants gave a positive score for clarity. A ratio between 55 % and 90 % gave an *agree* or *strongly agree*. The scenario receiving the most negative scores (21 %) was *variant synchronization*. From the textual feedback provided for this scenario, we observed that several participants struggled to understand a sub-scenario related to the classification of changes into either evolutionary or functional. For example, one participant stated that “*it is not entirely clear how an evolutionary change differs from a functional one.*” After we removed this sub-scenario and its description in the revision, we found that 86 % of the remaining participants gave a positive score. For the *transformation* scenario, we observed the same increase of positive scores (to 86 %) after we removed a sub-scenario related to the replacement of the used variability mechanism. For the other scenarios with comparatively many neutral or negative answers, we did not find any repeated issues occurring in the textual explanations.

**RQ<sub>2</sub>: Relevance.** A majority of participants (between 55 % and 95 %) assessed the relevance of each scenario positively. Interestingly, despite the lower scores for clarity, *variant synchronization* is among the two scenarios deemed relevant by 95 % of all participants. To study this discrepancy further, we analyzed the scores per sub-scenario. We found that most participants considered the sub-scenario that we removed in the revision (*classify changes*, 33 % positive responses) less relevant than the remaining *variant synchronization* sub-scenarios. Likewise, *transformations* attracted 100 % positive scores for overall relevance after we removed the least relevant sub-scenario (*exchange variability mechanism*, 33 % positive responses). In other cases with comparatively fewer positive scores (*architecture recovery* and *problem-solution space co-evolution*; 60 % and 63 % positive scores, respectively), it was not obvious from the textual comments how these scores can be explained. An interesting case is visualization. Despite the overall mid-range responses, two participants deemed it particularly relevant, but hard to benchmark: “*I believe visualization has much potential to improve many tasks in evolution of variant-rich systems. [...] Evaluation itself, in terms of measuring the impact, is harder.*”

The participants’ feedback confirms the clarity and relevance of our benchmark descriptions. The scenarios *variant synchronization*, *feature identification & location*, and *constraints extraction* were considered most relevant.



**Figure 12: Results of the survey concerning clarity and relevance for scenarios: Variant Synchronization, Feature Identification and Location, Constraints Extraction, Feature Model Synthesis, Variant Integration, Architecture Recovery, Functional Testing, Analysis of Non-Functional Properties, Visualization, and Co-Evolution of Problem & Solution Space.**

### 3.3 Threats to Validity

The external validity of our survey is threatened by the number of participants. However, since we focus on a highly specialized population—the community of variability and evolution experts—valid conclusions about that population can be supported by a smaller sample than a large population would require. By inviting the attendees of a relevant Dagstuhl seminar, we benefit from a pre-selection of experts in this area. Regarding conclusion validity, the confidence in our clarity scores could be improved by asking the participants to solve comprehension tasks, rather than having them rate the description clarity. However, such an experiment would have taken much more time and, therefore, would have risked to affect the completion rate.

## 4 SURVEYING EXISTING BENCHMARKS

In this section, we survey a selection of benchmarks with regard to their applicability to the scenarios we introduced in Sec. 2.

### 4.1 Methodology

**Selection.** As starting point for our selection of benchmarks, we collected a list of benchmarks that we were aware of, due to our familiarity with the field (convenience sampling). To get a more complete overview in a systematic way, we gathered additional benchmarks using a dedicated question in our community survey, in which we asked the participants to name benchmarks that they are aware of. Finally, since we found that a dedicated benchmark was not available for each scenario, we also considered benchmarks from related areas, such as *traceability* research, and identified whether they match our scenarios. From these steps, we derived an initial list of 17 benchmark candidates.

Based on our definition of benchmark, as given in Sec. 1, we defined the following inclusion criteria:

- I1 The availability of a dataset based on one or more systems created by industrial practitioners, and
- I2a The availability of a ground truth for assessing the correctness of a given technique, or
- I2b The availability of a framework for assessing other properties of interest.

From the initial 17 benchmark candidates, nine satisfied the inclusion criteria, meaning that they provided a suitable dataset, and either a ground truth or a framework for assessing a relevant technique. We focused on these nine benchmarks in our survey and excluded eight additional ones that did not satisfy all criteria. The excluded candidates can be considered as notable datasets, as they may still offer some value for benchmarking. We discuss the selected benchmarks in Sec. 4.2, and the notable datasets in Sec. 5.

**Assessment.** To determine how well our eleven scenarios are supported by the identified benchmarks and to identify synergies between benchmarks and scenarios, we assessed the suitability of each benchmark for each scenario. To this end, for a given benchmark candidate, we considered the requirements given in the benchmark descriptions (Sec. 2) and checked whether it fulfills the requirements and provides the artifacts that we defined.

### 4.2 Results

In Table 1, we provide an overview of the considered benchmarks and scenarios. The area from which the benchmark originally stems is given as *original context* in the table. A full circle indicates full support for at least one sub-scenario, a half-filled circle indicates partial support (i.e., a subset of the required artifacts is available) for at least one sub-scenario, and an empty circle indicates no support of the given scenario by means of the given benchmark. In the following, we briefly introduce the benchmarks and explain the cases in which a scenario is fully or partially supported.

**ArgoUML-SPL FLBench** [57] has a long tradition as benchmark for feature location in single systems and in families of systems [56]. The ground truth consists of feature locations for eight optional features of ArgoUML at the granularity of Java classes and methods. A feature model is available. The framework allows to generate predefined scenarios (a set of configurations representing a family) and to calculate metrics reports for a given feature location result. Given that this benchmark only contains eight optional features, we argue that it only partially satisfies the needs for feature location. **Drupal** [81] is a dataset of functional faults in the variable content management system Drupal. For each of the faults, it contains the feature or feature interaction responsible for triggering the fault. Moreover, the faults are reported over two different versions of

**Table 1: Mapping of existing benchmarks to scenarios, with circles indicating fulfillment of scenario requirements**

Benchmark	Original Context	VS	VI	FIL	CE	FMS	AR	TR	FT	ANF	VZ	CPS
<b>ArgoUML-SPL FLBench.</b> [57]	Feature location	○	○	●	○	○	○	○	○	○	○	○
<b>Drupal</b> [81]	Bug detection	○	○	○	○	○	○	○	●	○	●	○
<b>Eclipse FLBench</b> [63]	Feature location	○	○	●	●	●	○	○	○	○	○	○
<b>LinuxKernel FLBench.</b> [98]	Feature location	○	○	●	●	●	○	○	○	○	○	○
<b>Marlin &amp; BCWallet</b> [42]	Feature location	○	○	●	○	○	○	○	○	○	●	○
<b>ClaferWebTools</b> [37]	Traceability	○	○	●	○	●	○	○	○	○	●	○
<b>DoSC</b> [101]	Change discovery	○	●	●	○	●	○	●	○	○	●	○
<b>SystemsSwVarModels</b> [15]	FM synthesis	○	●	○	●	●	○	○	○	○	●	○
<b>TraceLab CoEST</b> [40]	Traceability	○	○	●	○	○	○	○	○	○	○	○
<b>Variability bug database</b> [1]	Bug detection	○	○	●	○	○	○	○	●	○	●	○

Drupal, which may indicate faults that were introduced by the evolution of the system. This dataset is useful for the scenario of functional testing, to evaluate whether the selected variants for application testing cover relevant feature interactions that are known to contain faults. Moreover, the information of feature interactions could be used to partially benchmark visualization.

**Eclipse FLBench** [63] is a benchmarking framework for feature location techniques in single systems and in families of systems. Since the ground-truth traces map features to components (i.e., Eclipse plugins), the granularity is coarse and there are no cross-cutting features, thus justifying only “requires” constraints, thus justifying partial support of feature location. This benchmark supports different Eclipse releases, each containing around 12 variants, 500 features, and 2,500 components. The Eclipse FLBench also contains information about feature dependencies and hierarchy, but only “requires” constraints, thus justifying partial support of constraints extraction and FM synthesis.

**Linux Kernel FL Bench** [98] is a database containing the ground-truth traces from a selection of features of the Linux Kernel to corresponding C code. It contains the locations of optional features within 12 product variants derived from three Linux kernel releases. For each variant, we have around 2,400 features and 160,000 ground-truth links between features and code units. The database contains information about “requires” and “excludes” feature constraints, as well as the feature model hierarchy, making it a suitable ground truth for constraints extraction and feature-model synthesis. However, as it was not its intended usage, more complex feature constraints are not captured.

**Marlin & BCWallet** [42] is a dataset of feature locations (represented as embedded code annotations), feature models, and feature fact sheets of two open-source systems, all of which can serve as ground truth for feature identification and location techniques. It comprises both mandatory and optional features. The annotations can also serve as input for feature dashboards that provide visualizations with several metrics [5], for instance, assets related to a feature, scattering degrees, and developers associated with each feature.

**ClaferWebTools** [37] is a dataset with feature locations of both mandatory and optional features, as well as feature models, together with an evolution history. ClaferWebTools is a clone & own-based system that evolved in four variants. Like Marlin & BCWallet, the locations are embedded into the source code. It can be used to evaluate feature-location techniques exploiting historical information, or visualization techniques showing the evolution of features.

**DoSC** (*Detection of Semantic Changes* [101]) is a dataset with revision histories of eight Java projects for benchmarking semantic change detection tools. Semantic changes are commits that correspond to entries from an issue tracking system; they can be considered as features in a broader sense. Traces from semantic changes to implementation elements are included, thus providing a ground truth for feature location (partially supported, since only optional features are considered) and a basis for visualization. The revision histories also provide a rich data source for benchmarking transformation and variant integration. However, full support is prohibited by the lack of a feature model and available ground truths.

**SystemsSwVarModels** [15] comprises a corpus of 128 extracted real-world variability models from open-source systems-software, such as the Linux kernel, the eCos operating system, BusyBox, and 12 others. The models are represented in the variability modeling languages Kconfig [85] and CDL [14], with the benchmark providing tools to analyze and transform these models into their configuration space semantics (expressed as Boolean, arithmetic, and string constraints), abstracted as propositional logics formulas. As such, these formulas can be used to benchmark constraints extraction from codebases and feature model synthesizes. To some extent, the corpus can be used to benchmark feature-oriented visualizations (e.g., slicing feature models) and problem & solution space co-evolution.

**TraceLab CoEST** [40] is an initiative of the Center of Excellence for Software and Systems Traceability gathering a set of case studies on traceability recovery with their corresponding ground-truth traces. We can find benchmarks with traces from requirements to source code, from requirements to components, from high- to low-level requirements, from use cases to source code, and other types of traces that partially satisfy the needs of evaluating feature location techniques in single systems.

**Variability Bug Database** [1] is an online database of 98 variability-related bugs in four open-source repositories: The Linux kernel, BusyBox, Marlin, and Apache. The meta-data provided for bug entries include a description, a type (e.g., “expected behavior violation”), a configuration, and pointers to a revision where the bug appears and where it is fixed. This database is especially useful for functional testing, as it provides a ground truth in the form of faults together with the configurations in which they appear. The projects contain #ifdef directives that can be considered as

variability annotations, rendering the database partially suitable for benchmarking feature location and visualization.

While we identified synergies between the scenarios and existing benchmarks, the overall coverage is still low: A complete benchmark is only available for three of the eleven considered techniques. Four scenarios lack any benchmark: variant synchronization, analysis of non-functional properties, architecture recovery, and co-evolution of problem & solution space. The former two were deemed as particularly relevant in our community survey.

## 5 RELATED WORK

Besides the benchmarks we analyzed in the previous section, we are aware of several datasets and proposals that aim to achieve similar ideas, and benchmarks from related areas.

**Repositories.** Some repositories collect artifacts or full projects in the domain of software-product-line engineering. For example, `spl2go` (<http://spl2go.cs.ovgu.de/>) provides a set of various software product lines. However, most of these systems are based on student projects and they provide solely downloadable code. A more extensive overview especially on extractive software-product-line adoption is offered by the ESPLA catalog [56]. ESPLA collects information from existing papers, rather than providing data or an infrastructure by itself. Similarly, tools like FeatureIDE [65] and PEoPL [9] provide some complete example product lines, but are neither industrial nor do they have ground truths.

**Case Studies and Datasets.** Some case studies have been introduced that partly aimed to provide the basis for establishing benchmarks. The potentially best-known and first of such studies is the graph product line introduced by Lopez-Herrejon and Batory [52]. McGregor [64] reports experiences of using the fictional arcade product line in teaching, but focuses solely on reporting established practices. Recently, several case studies have been reported that particularly aim to provide suitable data sets for evaluating techniques for the evolution and extraction for software product lines. For example, Martinez et al. [59] extracted a software product line from educational robotic variants. The Apo-Games [46] are a set of real-world games, realized using clone & own, with which the authors aim to provide a benchmark for the extraction of software product lines based on community contributions. Two recent works in fact provide datasets detailing the migration of dedicated subsets of the cloned games into product line platforms [4, 21]. BeTTy [83] is a feature model generator, focused on benchmarking and testing automated analysis techniques for feature models. Tzoref-Brill and Maoz [96] provide a dataset for assessing co-evolution techniques for combinatorial models and tests. A combinatorial model, similar to a configuration, is a set of bindings of parameters to concrete values. Finally, SPLIT [66] provides a set of 1,073 feature models and constraints, also including an editor and analysis framework. It mostly includes academic feature models and toy examples. None of these works represent a benchmark according to our criteria, namely that they are based on assets created by practitioners *and* provided together with a ground truth or assessment framework.

**Benchmarks in Related Areas.** Various benchmarks have been proposed in areas that are closely related to variability engineering.

SAT solvers are often applied in the context of software variability, specially for family-based analyses. The annual SAT competitions [36] provide various benchmarks and are important enablers for the SAT community. In the area of software-language engineering, the language workbench challenge [28] is an annual contest with the goal of promoting knowledge exchange on language workbenches. Model transformations provide the capability to represent product composition as transformation problem and have several established benchmarks, for instance, on graph transformation [97] and scalable model transformations [93]. While these benchmarks are complementary to the ones we consider in this paper, they report best practices that should be applied when implementing our envisioned benchmark set.

## 6 CONCLUSION AND ROADMAP

In this paper, we aimed to pave the way for a consolidated set of benchmark for techniques that support developers during the evolution of variant-rich systems. We studied relevant scenarios, investigated the clarity and relevance of the scenarios in a survey with variability and evolution experts, and surveyed the state of the art in benchmarking of these techniques.

**Results.** In summary, our main results are:

- We identified 11 scenarios covering the evolution of variant-rich systems, together with requirements for benchmarking the relevant techniques.
- Community feedback shows that our scenarios are clearly defined and important to advance benchmarking in the area.
- Only three out of the 11 scenarios are completely supported by existing benchmarks, highlighting the need for a consolidated benchmark set with full support for all scenarios.

**Roadmap.** Our results suggest the following research roadmap to eventually achieve such an envisioned benchmark set.

As a key goal, we aim to set up a common infrastructure for all scenarios presented in this paper. This way, we can utilize synergies between benchmarks, specifically by means of shared datasets and assets. Where available, we may reuse publicly available implementations of benchmark frameworks and integrate them.

Most scenarios require a manually curated ground truth. Therefore, creating ground truths for the available datasets is a substantial effort, a call for investing more resources into benchmarking. A further important goal is to broaden the scope of datasets. Most available datasets are based on open-source projects from traditional embedded systems. It is worthwhile to include datasets from upcoming domains that need variability handling, including data-analytics software [30], service robotics [31], and cyber-physical systems [16].

Raising awareness for the challenges and opportunities for benchmarking takes a concerted effort. Developers of new techniques shall be encouraged to use our benchmark infrastructure, articulate gaps in the benchmark literature, and fill them by contributing their own benchmarks. We plan to advertise our initiative in the appropriate mailing lists and social media.

**Acknowledgments.** Supported by ITEA project REVaMP<sup>2</sup> funded by Vinnova Sweden (2016-02804). We thank the participants of Dagstuhl seminar 19191, all survey participants, and Tewfik Ziadi for input and comments on earlier versions of this paper.

## REFERENCES

- [1] Iago Abal, Jean Melo, Stefan Stănculescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *ACM Transactions on Software Engineering and Methodology* 26, 3 (2018), 10:1–10:34.
- [2] Hadil Abukwaik, Andreas Burger, Berima Andam, and Thorsten Berger. 2018. Semi-Automated Feature Traceability with Embedded Annotations. In *ICSME*. IEEE.
- [3] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. 2012. On Extracting Feature Models from Product Descriptions. In *VaMoS*. ACM.
- [4] Jonas Akesson, Sebastian Nilsson, Jacob Krüger, and Thorsten Berger. 2019. Migrating the Android Apo-Games into an Annotation-Based Software Product Line. In *23rd International Systems and Software Product Line Conference (SPLC), Challenge Track*.
- [5] Berima Andam, Andreas Burger, Thorsten Berger, and Michel R. V. Chaudron. 2017. Florida: Feature Location Dashboard for Extracting and Visualizing Feature Traces. In *VaMoS*. ACM.
- [6] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleit, Ralf Lämmel, EY Stefan Stănculescu, Andrzej Wąsowski, and Ina Schaefer. 2014. Flexible Product Line Engineering with a Virtual Platform. In *ICSE*. ACM.
- [7] Patrizia Asirelli, Maurice H. Ter Beek, Alessandro Fantechi, and Stefania Gnesi. 2010. A Logical Framework to Deal with Variability. In *IFM*. Springer.
- [8] Wesley K. G. Assunção, Roberto E Lopez-Herrejon, Lukas Linsbauer, Silvia R Vergilio, and Alexander Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering* 22, 6 (2017), 2972–3016.
- [9] Benjamin Behringer, Jochen Palz, and Thorsten Berger. 2017. PEoPL: Projectional Editing of Product Lines. In *ICSE*. IEEE.
- [10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–636.
- [11] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. In *CAiSE*. Springer.
- [12] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *SPLC*. ACM.
- [13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *VaMoS*. ACM.
- [14] Thorsten Berger and Steven She. 2010. *Formal Semantics of the CDL Language*. Technical Report. Department of Computer Science, University of Leipzig, Germany.
- [15] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1611–1640.
- [16] Hamid Mirzaei Buini, Steffen Peter, and Tony Givargis. 2015. Including variability of physical models into the design automation of cyber-physical systems. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*. ACM, 1–6.
- [17] John Businge, Openja Moses, Sarah Nadi, Engineer Bainomugisha, and Thorsten Berger. 2018. Clone-Based Variability Management in the Android Ecosystem. In *ICSM*. IEEE.
- [18] Daniel Cruz, Eduardo Figueiredo, and Jabier Martinez. 2019. A Literature Review and Comparison of Three Feature Location Techniques using ArgoUML-SPL. In *VaMoS*. ACM.
- [19] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *VaMoS*. ACM.
- [20] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John D. McGregor, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. 2011. A Systematic Mapping Study of Software Product Lines Testing. *Information and Software Technology* 53, 5 (2011), 407–423.
- [21] Jamie Debbiche, Oskar Lignell, Jacob Krüger, and Thorsten Berger. 2019. Migrating the Java-Based Apo-Games into a Composition-Based Software Product Line. In *23rd International Systems and Software Product Line Conference (SPLC), Challenge Track*.
- [22] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. A Robust Approach for Variability Extraction from the Linux Build System. In *SPLC*. ACM.
- [23] Ivan do Carmo Machado, John D. McGregor, Yguarata Cerqueira Cavalcanti, and Eduardo Santana de Almeida. 2014. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *Information and Software Technology* 56, 10 (2014), 1183–1199.
- [24] Ivan do Carmo Machado, John D. McGregor, and Eduardo Santana de Almeida. 2012. Strategies for Testing Products in Software Product Lines. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–8.
- [25] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *CSMR*. IEEE.
- [26] Wolfgang Eixelsberger, Michaela Ogris, Harald Gall, and Berndt Bellay. 1998. Software Architecture Recovery of a Program Family. In *ICSE*. IEEE.
- [27] Sina Entekhabi, Anton Solback, Jan-Philipp Steghöfer, and Thorsten Berger. 2019. Visualization of Feature Locations with the Tool FeatureDashboard. In *23rd International Systems and Software Product Line Conference (SPLC), Tools Track*.
- [28] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriel P. Konat, Pedro J. Molina, Martin Palatinus, Risto Pohjonen, Eugen Schindler, Clemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches. In *SLE*. Springer.
- [29] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *ICSM*. IEEE.
- [30] Amir Gandomi and Murtaza Haider. 2015. Beyond the hype: Big data concepts, methods, and analytics. *International Journal of Information Management* 35, 2 (2015), 137–144.
- [31] Sergio García, Daniel Strüber, Davide Brugali, Alessandro Di Fava, Philipp Schillinger, Patrizio Pelliccione, and Thorsten Berger. 2019. Variability Modeling of Service Robots: Experiences and Challenges. In *VaMoS*. ACM, 8:1–6.
- [32] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. 2011. A Genetic Algorithm for Optimized Feature Selection with Resource Constraints in Software Product Lines. *Journal of Systems and Software* 84, 12 (2011), 2208–2221.
- [33] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Attrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wąsowski, and Huiqun Yu. 2018. Data-Efficient Performance Learning for Configurable Systems. *Empirical Software Engineering* 23, 3 (2018), 1826–1867.
- [34] Haitham S. Hamza, Jabier Martinez, and Carmen Alonso. 2010. Introducing Product Line Architectures in the ERP Industry: Challenges and Lessons Learned. In *SPLC*.
- [35] Daniel Hinterreiter, Herbert Prähofer, Lukas Linsbauer, Paul Grünbacher, Florian Reisinger, and Alexander Egyed. 2018. Feature-Oriented Evolution of Automation Software Systems in Industrial Software Ecosystems. In *ETFA*. IEEE.
- [36] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. 2012. The International SAT Solver Competitions. *AI Magazine* 33, 1 (2012), 89–92.
- [37] Wenbin Ji, Thorsten Berger, Michał Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability with Embedded Annotations. In *SPLC*. ACM.
- [38] Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Carnegie-Mellon University, Pittsburgh, PA, USA.
- [39] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. 2014. Variability Mining: Consistent Semi-Automatic Detection of Product-Line Features. *IEEE Transactions on Software Engineering* 40, 1 (2014), 67–82.
- [40] Ed Keenan, Adam Czuderna, Greg Leach, Jane Cleland-Huang, Yonghee Shin, Evan Moritz, Malcom Gethers, Denys Poshyvanyk, Jonathan Maletic, Jane Huffman Hayes, Alex Dekhtyar, Daria Manukian, Shervin Hosseini, and Derek Hearn. 2012. TraceLab: An Experimental Workbench for Equipping Researchers to Innovate, Synthesize, and Comparatively Evaluate Traceability Solutions. In *ICSE*. IEEE.
- [41] Rainer Koschke, Pierre Frenzel, Andreas P. J. Breu, and Karsten Angstmann. 2009. Extending the Reflexion Method for Consolidating Software Variants into Product Lines. *Software Quality Journal* 17, 4 (2009), 331–366.
- [42] Jacob Kräijer, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is my Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software* 152 (2019), 239–253.
- [43] Charles W. Krueger. 2002. Easing the Transition to Software Mass Customization. In *SPFE*. Springer.
- [44] Jacob Krüger, Mustafa Al-Hajjaji, Sandro Schulze, Gunter Saake, and Thomas Leich. 2018. Towards Automated Test Refactoring for Software Product Lines. In *SPLC*. ACM.
- [45] Jacob Krüger, Thorsten Berger, and Thomas Leich. 2019. *Software Engineering for Variability Intensive Systems*. CRC Press, Chapter Features and How to Find Them: A Survey of Manual Feature Location, 153–172.
- [46] Jacob Krüger, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. 2018. Apo-Games: A Case Study for Reverse Engineering Variability from Cloned Java Variants. In *SPLC*. ACM.
- [47] Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2017. Finding Lost Features in Cloned Systems. In *SPLC*. ACM.
- [48] Miguel A. Laguna and Yania Crespo. 2013. A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product

- Line Refactoring. *Science of Computer Programming* 78, 8 (2013), 1010–1034.
- [49] Jihyun Lee, Sungwon Kang, and Danhyung Lee. 2012. A Survey on Software Product Line Testing. In *SPLC*. ACM.
- [50] Max Lillack, Stefan Stănicălescu, Wilhelm Hedman, Thorsten Berger, and Andrzej Wąsowski. 2019. Intention-Based Integration of Software Variants. In *ICSE*. ACM.
- [51] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *GPCE*. ACM.
- [52] Roberto E. Lopez-Herrejon and Don Batory. 2001. A Standard Problem for Evaluating Product-Line Methodologies. In *GPCE*. Springer.
- [53] Roberto E. Lopez-Herrejon and Alexander Egyed. 2013. Towards Interactive Visualization Support for Pairwise Testing Software Product Lines. In *VISOFT*. IEEE.
- [54] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux Kernel Variability Model. In *SPLC*. Springer.
- [55] Levi Lúcio, Moussa Amrani, Jürgen Dingel, Leen Lambers, Rick Salay, Gehan M. K. Selim, Eugene Syriani, and Manuel Wimmer. 2016. Model Transformation Intent and Their Properties. *Software and Systems Modeling* 15, 3 (2016), 647–684.
- [56] Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. 2017. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *SPLC*. ACM.
- [57] Jabier Martinez, Nicolas Ordonez, Xhevahire Ternava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. 2018. Feature Location Benchmark with ArgoUML SPL. In *SPLC*. ACM.
- [58] Jabier Martinez, Jean-Sébastien Sotter, Alfonso García Frey, Tewfik Ziadi, Tegawendé F. Bissyandé, Jean Vanderdonckt, Jacques Klein, and Yves Le Traon. 2017. Variability Management and Assessment for User Interface Design. In *Human Centered Software Product Lines*. Springer.
- [59] Jabier Martinez, Xhevahire Ternava, and Tewfik Ziadi. 2018. Software Product Line Extraction from Variability-Rich Systems: The Robocode Case Study. In *SPLC*. ACM.
- [60] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Bottom-Up Adoption of Software Product Lines: A Generic and Extensible Approach. In *SPLC*. ACM.
- [61] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Name Suggestions During Feature Identification: The Variclouds Approach. In *SPLC*. ACM.
- [62] Jabier Martinez, Tewfik Ziadi, Raul Mazo, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2014. Feature Relations Graphs: A Visualisation Paradigm for Feature Constraints in Software Product Lines. In *VISOFT*. IEEE.
- [63] Jabier Martinez, Tewfik Ziadi, Mike Papadakis, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2018. Feature Location Benchmark for Extractive Software Product Line Adoption Research Using Realistic and Synthetic Eclipse Variants. *Information and Software Technology* 104 (2018), 46–59.
- [64] John D. McGregor. 2014. Ten Years of the Arcade Game Maker Pedagogical Product Line. In *SPLC*. ACM.
- [65] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.
- [66] Marcilio Mendonça, Moises Branco, and Donald Cowan. 2009. SPLIT: Software Product Lines Online Tools. In *OOPSLA*. ACM.
- [67] Mukelabai Mukelabai, Damir Nešić, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *ASE*. ACM.
- [68] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining Configuration Constraints: Static Analyses and Empirical Results. In *ICSE*. ACM.
- [69] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering* 41, 8 (2015), 820–841.
- [70] Sana Ben Nasr, Guillaume Bécan, Mathieu Acher, João Bosco Ferreira Filho, Nicolas Sannier, Benoit Baudry, and Jean-Marc Davril. 2017. Automated Extraction of Product Comparison Matrices from Informal Product Descriptions. *Journal of Systems and Software* 124 (2017), 82–103.
- [71] Daren Nestor, Steffen Thiel, Goetz Botterweck, Ciarán Cawley, and Patrick Healy. 2008. Applying Visualisation Techniques in Software Product Lines. In *SoftVis*. ACM.
- [72] Lais Neves, Leopoldo Teixeira, Demóstenes Sena, Václav Alves, Uirá Kulezsa, and Paulo Borba. 2011. Investigating the Safe Evolution of Software Product Lines. *ACM SIGPLAN Notices* 47, 3 (2011), 33–42.
- [73] Leonardo Passos, Jesús Padilla, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Marco Tulio Valente. 2015. Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In *MODULARITY*. ACM.
- [74] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Padilla. 2018. A Study of Feature Scattering in the Linux Kernel. *IEEE Transactions on Software Engineering* (2018).
- [75] Ken Peffers, Tuire Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. 2007. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems* 24, 3 (2007), 45–77.
- [76] Juliana Alves Pereira, Jabier Martinez, Hari Kumar Gurudu, Sebastian Krieter, and Gunter Saake. 2018. Visual Guidance for Product Line Configuration using Recommendations and Non-Functional Properties. In *SAC*. ACM.
- [77] Tristan Pfoff, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. 2016. Synchronizing Software Variants with VariantSync. In *SPLC*. ACM.
- [78] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2013. Managing Cloned Variants: A Framework and Experience. In *SPLC*. ACM.
- [79] Andrey Sadovyykh, Tewfik Ziadi, Jacques Robin, Elena Gallego, Jan-Philipp Steghöfer, Thorsten Berger, Alessandra Bagnato, and Raul Mazo. 2019. RE-VAMP2 Project: Towards Round-Trip Engineering of Software Product Lines – Approach, Intermediate Results and Challenges. In *TOOLS 50 +*.
- [80] Rick Salay, Michalis Famelis, Julia Rubin, Alessio Di Sandro, and Marsha Chechik. 2014. Lifting Model Transformations to Product Lines. In *ICSE*. ACM.
- [81] Ana B. Sánchez, Sergio Segura, José Antonio Parejo, and Antonio Ruiz Cortés. 2017. Variability Testing in the Wild: The Drupal Case Study. *Software and System Modeling* 16, 1 (2017), 173–194.
- [82] Sandro Schulze, Thomas Thüm, Martin Kuhlemann, and Gunter Saake. 2012. Variant-Preserving Refactoring in Feature-Oriented Software Product Lines. In *VaMoS*. ACM.
- [83] Sergio Segura, José A. Galindo, David Benavides, José Antonio Parejo, and Antonio Ruiz Cortés. 2012. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In *VaMoS*. ACM.
- [84] Anas Shatnawi, Abdelhak-Djamel Serai, and Houari Sahraoui. 2017. Recovering Software Product Line Architecture of a Family of Object-Oriented Product Variants. *Journal of Systems and Software* 131 (2017), 325–346.
- [85] Steven She and Thorsten Berger. 2010. *Formal Semantics of the Kconfig Language*. Technical Report. Electrical and Computer Engineering, University of Waterloo, Canada.
- [86] Steven She, Krzysztof Czarnecki, and Andrzej Wąsowski. 2012. Usage Scenarios for Feature Model Synthesis. In *VARY*. ACM.
- [87] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *ICSE*. ACM.
- [88] Steven She, Uwe Ryssel, Nele Andersen, Andrzej Wąsowski, and Krzysztof Czarnecki. 2014. Efficient Synthesis of Feature Models. *Information and Software Technology* 56, 9 (2014), 1122–1143.
- [89] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-Influence Models for Highly Configurable Systems. In *ESEC/FSE*. ACM.
- [90] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward Optimization of Non-Functional Properties in Software Product Lines. *Software Quality Journal* 20, 3–4 (2012), 487–517.
- [91] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. 2003. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In *ICSE*. IEEE.
- [92] Zipani Tom Sinkala, Martin Blom, and Sebastian Herold. 2018. A Mapping Study of Software Architecture Recovery for Software Product Lines. In *ECSA*. ACM.
- [93] Daniel Strüber, Timo Kehrer, Thorsten Arendt, Christopher Pietsch, and Dennis Reuling. 2016. Scalability of Model Transformations: Position Paper and Benchmark Set. In *BigMDE*.
- [94] Stefan Stănicălescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *ICSM*. IEEE.
- [95] Pablo Trinidad, David Benavides, Antonio Ruiz-Cortés, Sergio Segura, and Alberto Jimenez. 2008. FAMA Framework. In *SPLC*. IEEE.
- [96] Rachel Tzoref-Brill and Shahar Maoz. 2018. Modify, Enhance, Select: Co-Evolution of Combinatorial Models and Test Plans. In *ESEC/FSE*. ACM.
- [97] Gergely Varró, Andy Schürr, and Dániel Varró. 2005. Benchmarking for Graph Transformation. In *VL/HCC*. IEEE.
- [98] Zhenchang Xing, Yinxing Xue, and Stan Jarzabek. 2013. A Large Scale Linux-Kernel based Benchmark for Feature Location Research. In *ICSE*. ACM.
- [99] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. 2012. Feature Location in a Collection of Product Variants. In *WCSE*. IEEE.
- [100] Shurui Zhou, Stefan Stănicălescu, Olaf Lefebvre, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. 2018. Identifying Features in Forks. In *ICSE*. IEEE.
- [101] Chenguang Zhu, Yi Li, Julia Rubin, and Marsha Chechik. 2017. A Dataset for Dynamic Discovery of Semantic Changes in Version Controlled Software Histories. In *MSR*. IEEE.

# Industrial and Academic Software Product Line Research at SPLC: Perceptions of the Community\*

Rick Rabiser

CDL MEVSS

Johannes Kepler University  
Linz, Austria  
rick.rabiser@jku.at

Klaus Schmid

University of Hildesheim  
Hildesheim, Germany  
schmid@sse.uni-hildesheim.de

Martin Becker

Fraunhofer IESE  
Kaiserslautern, Germany

martin.becker@iese.fraunhofer.de  
Goetz Botterweck

Lero, University of  
Limerick

Limerick, Ireland  
goetz.botterweck@lero.ie

Matthias Galster

University of Canterbury  
Christchurch, New Zealand  
mgalster@ieee.org

Iris Groher

Johannes Kepler University  
Linz, Austria  
iris.groher@jku.at

Danny Weyns

KU Leuven, Belgium  
Linnaeus Univ., Sweden  
danny.weyns@gmail.com

## ABSTRACT

We present preliminary insights into the perception of researchers and practitioners of the software product line (SPL) community on previous, current, and future research efforts. We were particularly interested in up-and-coming and outdated topics and whether the views of academics and industry researchers differ. Also, we compared the views of the community with the results of an earlier literature survey published at SPLC 2018. We conducted a questionnaire-based survey with attendees of SPLC 2018. We received 33 responses (about a third of the attendees) from both, very experienced attendees and younger researchers, and from academics as well as industry researchers. We report preliminary findings regarding popular and unpopular SPL topics, topics requiring further work, and industry versus academic researchers' views. Differences between academic and industry researchers become visible only when analyzing comments on open questions. Most importantly, while topics popular among respondents are also popular in the literature, topics respondents think require further work have often already been well researched. We conclude that the SPL community needs to do a better job preserving and communicating existing knowledge and particularly also needs to widen its scope.

## CCS CONCEPTS

• Software and its engineering → *Software product lines*.

## KEYWORDS

Software product lines, industry, academia, SPLC

\*The first author coordinated the efforts presented in this paper; the second author created the initial concept; all other authors contributed equally/are listed alphabetically.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336310>

## ACM Reference Format:

Rick Rabiser, Klaus Schmid, Martin Becker, Goetz Botterweck, Matthias Galster, Iris Groher, and Danny Weyns. 2019. Industrial and Academic Software Product Line Research at SPLC: Perceptions of the Community. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3336294.3336310>

## 1 INTRODUCTION

Various systematic literature reviews and surveys show the large body of research that has been published on diverse topics related to SPLs [1, 2, 4, 6, 9, 11–19, 24, 25, 27, 28]. Over time, researchers from academia and industry contributed numerous papers: more than 600 at the Software Product Line Conference (SPLC), many more at other venues like ICSE, ESEC/FSE, ESEM, WICSA, GPCE, ASE, VaMoS and journals, such as TSE, TOSEM, JSS, IST, as well as textbooks, edited books, and technical reports.

In an earlier study [26], we analyzed a subset (140) of all papers published at SPLC from 1996 to 2017 to determine whether there is an (increasing) gap between research conducted in academia and industry. We assessed the research type of the papers (academic or industry), the kind of evaluation (application example, empirical, etc.), and the application domain. Also, we determined the SPL life-cycle phases, development practices, and topics the papers address and how they changed over time. We concluded that even though several topics have received more attention than others, academic and industry research on SPLs overall seem to be actually rather in line with each other when just looking at the literature.

In contrast to the results from our literature study, for us and several colleagues we talked to, it still seems that research conducted today in industry and academia is quite different and (in contrast to the early days of SPL research) increasingly does not align well. This difference in our perception might be explained by the fact that research publications usually only report “success stories”, that we only analyzed 140 of all SPL-related publications, and that it is simply difficult to get a clear picture by just looking at publications. Thus, to either back up or refute our earlier results, we decided to ask the SPL community, i.e., find out what the perceptions of academics and industry researchers in the field of SPLs are.

Here, we report our findings from a questionnaire-based survey we conducted at SPLC 2018, aiming at two research questions.

**RQ1:** What are *up-and-coming* and what are *well-studied* SPL research topics? Are there *differences when distinguishing academic researchers and industry researchers?*

**RQ2:** Do responses on *topics* given in the survey *differ from those in publications* investigated in our earlier study [26]?

We received responses from about a third of the 102 participants of the SPL conference, i.e., 33, from very experienced researchers as well as young PhD students and from academic researchers as well as industry researchers. *Disclaimer:* Due to the small sample size (33), we mainly present ranking-based comparisons (top N in one category vs. another category). In general, our results have to be viewed as preliminary and qualitative in nature (with only some, mainly relative numbers).

## 2 RESEARCH METHOD

We conducted a questionnaire-based survey following the guidelines by Ciolkowski et al. [10]. Our **target population** included SPL researchers and practitioners (asking respondents as part of the survey, what they consider themselves). We call researchers academic researchers (short: AR) and practitioners industry researchers (short: IR) here, as all attendees of SPLC can be viewed as researchers, even if they work in or close to industry. Please note that maybe not all industry participants would consider themselves researchers, but they are at least interested in SPL research (results) or otherwise they would not attend SPLC.

For **sampling**, we used purposive sampling [30], because respondents needed to be actively involved in the SPL community. To recruit participants, we advertised the survey at SPLC 2018.

For **data preparation and collection**, we used a self-administered online questionnaire<sup>1</sup>, distributed as a Google form. The raw data can be found online<sup>2</sup>. The questionnaire was reviewed by others from the target population not involved in the research. It was also evaluated through a series of pilots with representatives from the target population and revised accordingly. Questions are structured in three main parts. Part 1 collects demographic information about respondents such as whether they consider themselves an (academic) researcher or (industry) practitioner and what their experience with SPL research and SPLC is. Part 2 comprises an open question on their opinion regarding SPL topics, which should be investigated (more), before presenting a list of topics—(based on our topic list from [26], derived from existing SPL frameworks, such as [29])—asking respondents to mark for each topic whether they think it has been well investigated by the SPL community, whether it has so far not been investigated (enough), and whether they are personally interested in the topic. In Part 3, we ask respondents what their general impression regarding the alignment of topics investigated in industry vs. in academia at SPLC is, using several open questions. We also ask them on their perception of the impact of current developments in systems and software engineering (e.g., Cloud, DevOps, Cyber-physical systems, IoT, Big Data, etc.) on the

SPL community to identify future directions of the field and allow them to add any further general comments.

For **data analysis**, we used descriptive statistics and quantitative analysis to provide answers to our research questions:

Regarding RQ1, we analyzed responses to the question “In your opinion, what SPL topics should be investigated (more)?” Seven of the ten industry researchers and 16 of the 23 academic researchers provided answers to this question. Answers to this question were given as free-form text. Thus, we applied open coding in order to identify topics. Two researchers coded the answers independently and results were discussed with all researchers. Further, we analyzed the answers to a closed question (“Please mark those topics that are currently (or have been) well investigated by the SPL community and those topics that have so far not been investigated (enough) by the SPL community. Also, please indicate in which topics you are personally interested in.”) using descriptive statistics. All respondents answered that question. We also compared AR vs. IR responses when looking at the responses to these questions.

Regarding RQ2, we compared the results from RQ1 with the results of our earlier literature study [26]. We referred to the most recent publications to find out whether there are topics that people would like to see that are not much covered in the literature in the last years. In addition to these research questions, we analyzed answers to the open questions regarding respondents’ general impressions on academic vs. industry research, current developments, and other comments.

## 3 RESULTS

Ten respondents consider themselves practitioners (IR) and 23 consider themselves academics (AR). Ten are still PhD students (one in industry). The average experience with SPLs/SPL research our respondents have is 8.6 years (ARs: 7.3, IRs: 10.8; min: 0, max: 24). SPLs are the main research topic for 20 of our 33 respondents, 15 ARs and five IRs. Of 22 total SPLC conferences/workshops 1996–2018, respondents on average attended five events (min: 1, max: 21). There is no real difference between ARs and IRs here. 76% (19 ARs and six IRs) have published at least one paper at SPLC. IRs publish more in the industry track (six of ten), but four respondents also have published papers in the research track. ARs publish more in the research track (18 of 23), but eight have also published industry track papers. Interestingly, 18 people (15 ARs and three IRs) have at least once attended an SPLC before without having a paper.

### 3.1 RQ1: Popular and Unpopular Topics

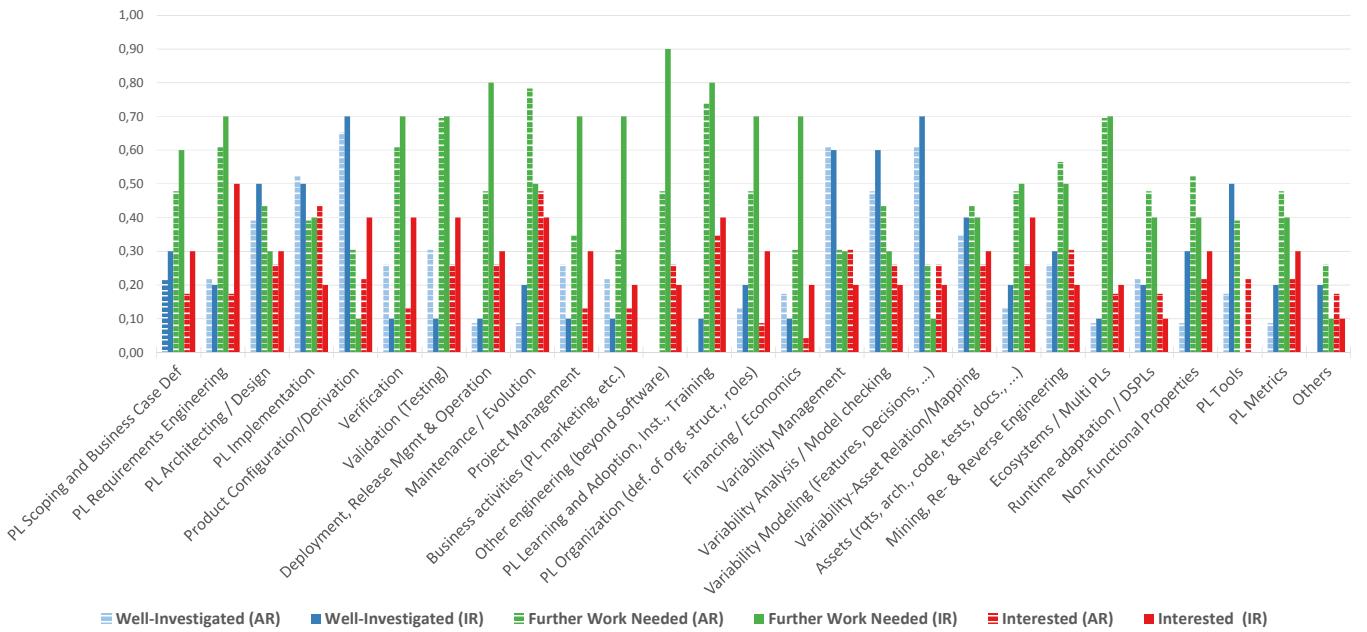
Figure 1 shows what topics respondents (AR vs. IR) found well-investigated vs. think require future work and also what topics they are personally interested in.

**3.1.1 Well-studied topics and topics that need to be investigated more.** Based on closed question “Please mark those topics …”, we identified the top three topics considered as “investigated well”, “requires further work”, and “have an interest in” as shown in Table 1.

Regarding the open question on topics, due to the diversity of topics identified through coding textual answers (we identified 28 topics in the 23 responses), we were not able to identify topics that are clearly dominating regarding the need to study them further. Some topics appeared in more than one answer, i.e., security, testing,

<sup>1</sup><https://tinyurl.com/SPLCommSurv>

<sup>2</sup><https://tinyurl.com/SPLCommSurveyData>



**Figure 1: Relative numbers of topics selected as “are well-investigated”, “require future work”, and “am personally interested in” as part of closed question “Please mark those topics that...” by academic researchers (AR) vs. industry researchers (IR) in our survey. For each topic, on average about 28% of respondents didn’t provide a reply (min: 12%; max: 73%).**

evolution, organizational issues and formal methods. However, there was no clear trend. Most topics provided in the open question are also reflected in the options of the closed question.

**3.1.2 Differences between Academic and Industry Researchers.** Based on Figure 1 and the list of topics in Table 1, we could not identify any systematic differences between the topics considered more or less important between industry and academic researchers. Interestingly, IRs appear to consider variability modeling as more well covered compared to ARs. On the other hand, IRs consider maintenance and evolution not addressed as well as ARs.

An interesting finding is that SPL tools are more in the focus of ARs than of IRs, i.e., no IR mentioned any tool. 50% of IRs think SPL tools have been well-investigated, no IR responded s/he was interested in tools or thought any further work is needed. Only 17% of ARs think tools have been well-investigated, 39% think more work is needed, and 22% are interested in tools.

For the non-technical SPL engineering topics Project Management, Business activities, and Financing / Economics, there is a noticeable difference in the opposite direction: IRs (58%) show a stronger interest in further research than ARs (30%). Of similar nature is the extension of SPL engineering to other engineering disciplines beyond software. None of the respondents feels that the topic is well-investigated and 75% of IRs and 41% of ARs think further research is needed.

Deployment, Release management, and Operation is another topic where a bigger difference between the groups of respondents can be perceived: 62% of IRs in contrast to 39% of ARs have expressed a need for further research.

Regarding other topics, IRs did only mention one additional topic: ALM/PLM Integration. Academic researchers mentioned various additional topics as potentially interesting, e.g., 3D printing, generative adversarial networks, etc. (see raw data linked in Section 2).

### 3.2 RQ2: Perceptions versus Literature

In our previous work we investigated topics of papers published at SPLC [26]. Here, we aim to contrast these results with our survey results. To do so, we determined the top-five and the bottom-five topics that were mentioned in the categories *well investigated*, *further work needed*, and *interested* in our survey. For simplicity, we ignored for both the literature analysis and the survey discussed in this paper the difference between academic and industrial viewpoints. Table 2 summarizes our comparison: cells highlighted in green mean that survey and literature correspond well, orange means there are some differences, and red means there is a significant divergence.

The five topics given as most well-investigated topics in the survey are *Product Configuration/Derivation*, *Variability Modeling*, *Variability Management*, *Variability Analysis / Model checking*, and *Variability-Asset Relation/Mapping*. If we compare<sup>3</sup> this with the results from our literature analysis, we see that these topics are also ranking among the top topics in the literature. Hence, the well-investigated topics roughly correspond.

If we look at the bottom topics of the survey (those that survey participants think have been least well investigated; ignoring the item “others”), the picture is much more mixed. Topics like

<sup>3</sup>Please note that the categories were a bit more fine-grained in our literature study than in the survey

**Table 1: Top three topics for categories well-investigated, further work needed (three 2nd and two 3rd places), and interested in (two 2nd and two 3rd places) based on replies by academic researchers (AR) and industry researchers (IR).**

Top Well-investigated				Top Further Work Needed				Top Interested			
Topic	Resp.	AR	IR	Topic	Resp.	AR	IR	Topic	Resp.	AR	IR
1. Prod. Conf./Derivation	22 (67%)	15 (66%)	7 (70%)	1. PL Learn., Adopt., Inst., Train.	25 (76%)	17 (74%)	8 (80%)	1. Maintenance / Evolution	15 (45%)	11 (48%)	4 (40%)
2. Variability Modeling	21 (63%)	14 (61%)	7 (70%)	2. Validation (Testing)	23 (70%)	16 (70%)	7 (70%)	2. PL Implementation	12 (36%)	10 (44%)	2 (20%)
3. Variability Management	20 (61%)	14 (61%)	6 (60%)	2. Maintenance / Evolution	23 (70%)	18 (78%)	5 (50%)	2. PL Learn., Adopt., Inst., Train.	12 (36%)	8 (35%)	4 (40%)
				3. Ecosystems / Multi PLs	23 (70%)	16 (70%)	7 (70%)	3. Validation (Testing)	10 (30%)	6 (26%)	4 (40%)
				3. PL Requirements Engineering	21 (64%)	14 (61%)	7 (70%)	3. Assets (reqts, arch, code, ...)	10 (30%)	6 (26%)	4 (40%)
				3. Verification	21 (64%)	14 (61%)	7 (70%)				

**Table 2: Top and bottom five topics for categories well-investigated, further work needed, and interested as expressed by survey participants vs. literature survey (green (normal text): survey answers correspond with literature, orange (text in italics): there are some differences, red (text in bold italics) there is a large divergence).**

Top Well-investigated	Top Further Work Needed	Top Interested
Product Configuration/Derivation	<i>PL Learning and Adoption, Inst., Training</i>	<i>Maintenance / Evolution</i>
Variability Modeling	<i>Validation (Testing)</i>	<i>PL Implementation</i>
Variability Management	<i>Maintenance / Evolution</i>	<i>PL Learning and Adoption, Inst., Training</i>
Variability Analysis / Model checking	Ecosystems / Multi PLs	<i>Validation (Testing)</i>
Variability-Asset Relation/Mapping	<i>PL Requirements Engineering</i>	<i>Assets (reqts, arch, code, ...)</i>
Bottom Well-investigated	Bottom Further Work Needed	Bottom Interested
<i>PL Learning and Adoption, Inst., Training</i>	Others	<i>Financing / Economics</i>
Deployment, Release Mgmt & Operation	Variability Modeling	<i>Business Activities (PL Marketing, etc.)</i>
Ecosystems / Multi PLs	Product Configuration/Derivation	<i>PL Organization</i>
<b>Maintenance / Evolution</b>	PL Tools	<i>Runtime Adaptation / DSPLs</i>
<i>PL Metrics</i>	Variability Management	<i>PL Tools</i>

Ecosystems/Multi PLs and Deployment, Release Management & Operation have indeed received only little attention in the literature. For PL Learning and Adoption, Institutionalization, Training and PL Metrics the situation, however, is rather different. There exists a non-trivial number of papers—e.g., [5, 22]—but not so many have been published over the last five years. A very different situation exists for Maintenance / Evolution. Survey respondents see this as a not very well-investigated topic. On the other hand, there are 20 (total), respectively seven (last five years) papers on this topic we analyzed in our literature study. The topic can thus not be viewed as “rarely investigated”.

When looking at the survey answers in category *further work needed* we notice that several of the bottom topics in category *well investigated* correspond to the top topics in category *further work needed*, which would be expected. However, as discussed above and in our literature survey, for many of the topics, there exist already a very significant number of publications. For instance, the topic PL Requirements Engineering is listed as a top topic in category *further work needed*, but corresponds to overall 39 (six in the last five years) papers we analyzed in our literature survey.

The topics mentioned as bottom-five survey answers in category *further work needed* are topics that have indeed received a significant number of publications. Thus, it appears plausible for them to be regarded as sufficiently researched.

If we look at the category *interested*, we find a very interesting picture: the top topics survey respondents are interested in are topics, which indeed have already received a significant amount of research (e.g., Maintenance / Evolution, PL Implementation, etc.). On the other hand, topics for which people expressed the least interest are also strongly overlapping with those that received so far the least attention in publications (e.g., Financing/Economics, Business Activities, PL Organization or Runtime adaptation / DSPLs).

We can conclude that the overall understanding of *what has been well researched roughly corresponds to our literature analysis*. However, at the same time *quite a few topics are considered by survey respondents as needing further work, which also have already received a very significant amount of attention in the literature over the years* (only considering quantity, not quality). Also, survey respondents expressed a lot of interest on topics that have already been well researched, while they didn't express interest on topics that so far have received little attention. Interestingly, some of these topics also rank rather high (not top five, but top half of all) in category *further work needed*, e.g., Runtime Adaptation / DSPLs or PL Organization.

## 4 DISCUSSION

Regarding RQ1 (cf. Section 3.1), we noticed that mostly *popular (or “hot”)* topics in software engineering (SE) such as machine learning, AI, cloud computing, continuous development, etc. did not appear in the topics mentioned by respondents (only machine learning was mentioned by one AR). In general, the answers to RQ1 do not provide a clear direction where the SPL domain currently is or where it should be heading. This raises the question of whether there is a disconnect between what the SPL community is working on and what is considered important by the broader SE community. Also, it raises the question of why SPL (or SE) researchers do not submit work related to “hot” topics to venues like SPLC, particularly as there is quite some work on hot topics, e.g., machine learning, appearing in major SE conferences these days (see, for instance, the program of ICSE 2019<sup>4</sup>). Identifying such relations could be highly rejuvenating for the SPL community. Making an explicit push in terms of panels, keynotes or lightning talks on such current topics and their relation

<sup>4</sup><https://2019.icse-conferences.org/program/program-icse-2019>

to SPLs might encourage such work, ideally in research-industry cooperation.

Regarding RQ2, perceptions vs. the literature, our results suggest that *many researchers who participated in the survey may not be aware of older work on certain topics* such as PL Adoption and PL Requirements Engineering. For some topics such as Maintenance / Evolution there even exists recent work and still there is a divergence between the survey and the literature study. It may be that survey participants are simply not aware of the published work or that they still count it as little/not enough work, particularly for Maintenance / Evolution, which is indeed a huge field with many issues motivating future work [18]. On the one hand one could argue that people should just better research related work to find existing work before “re-inventing the wheel”. On the other hand, maybe the overall research community is doing a bad job preserving and communicating knowledge. Maybe publications are not enough and we need some more sophisticated methods such as those discussed in the area of systematic knowledge engineering [7] providing, e.g., semantic search queries and glossaries. For many areas thematic bibliography sites exist<sup>5</sup>. However, this might be very hard to do for SPL engineering due to the huge body of research in this area, which is distributed across a large number of different venues. It might also be useful, especially with respect to the older work, to have a broad textbook, which covers this. Some existing textbooks like [23, 29] are just too old, while newer ones like [3] do address a subset of aspects of SPL engineering. Another idea may also be to organize “retrospective talks”, i.e., invite a renowned speaker on a topic with a discrepancy between literature and perception.

We also noticed that the *interest in certain topics vs. the feeling that more research is needed on these topics diverges*. For example, 74% of ARs and 90% of IRs think more research on other engineering beyond software (e.g., systems engineering) is needed, while for the same topic only 26% of ARs and 20% of IRs are interested. Similarly, for ecosystems and multi product lines, about 70% of both, ARs and IRs, think more research is needed, while only 17% of ARs and 20% of IRs are interested in these topics. The same pattern can be found for most topics: on average, 48% of ARs and 47% of IRs think more research is needed on diverse topics, while only 23% of ARs and 27% of IRs are interested in doing such research. These might be among the toughest issues to address as they imply the need for interdisciplinary research initiatives. So far these did not happen, but they are needed, if these blind spots are to be resolved.

*Regarding a missing synchronization of topics between academia and industry, we noticed very different views on who is lagging behind.* One IR, for example, stated that “many of the academic papers are exploring areas that industry has already solved in practice”. In contrast, an AR mentioned that “methods adopted by industry seem to be about ten years behind the state of the art and a part of their problems could be addressed by using up-to-date methods”. This points to a large need for more exchange between industry and academia and more initiatives for applied research and academia-industry collaboration. However, we assume this is not specific to SPLs but true for software engineering at large [8].

Regarding other thoughts participants wanted to share, one participant stated that “the conference community has become smaller

<sup>5</sup>e.g., see <http://program-repair.org/bibliography.html> for research on program repair.

and more narrowly focused”. Another comment was that “the SPL acronym for me is associated with a too specific, maybe too narrow vision of a software engineering process”. While such comments are indicators of issues, it is difficult to say whether these are issues that are truly different from the ones identified earlier. Feeling that the conference is too narrowly focused may be due to a lack of awareness of the breadth of existing work. For example, the lack of awareness of work that addresses agile product lines [12, 20, 21] may contribute to the feeling that the SPL community has a too narrow vision of software engineering, but it may also be related to the current lack of work on currently hot topics at SPLC. Creating a better understanding of these issues requires a more detailed and broader follow-up study.

## 5 THREATS TO VALIDITY

Threats to *construct validity* are about the appropriateness of the measures used in the study. Although we reviewed and piloted the questionnaire, our respondents might have interpreted questions differently than intended. This could have led to misleading findings. Threats to *internal validity* are about confounding factors that could have impacted our results and causal relationships and the appropriateness of the conclusions drawn from our study. It is possible that we were biased in the interpretation of the answers. We reduced this threat by having multiple individuals involved in the evaluation of the data. Also, by giving answer options to respondents for most questions, we did not need to determine what respondents might have meant in their answer. *External validity* is about the generalization of our findings. Our sample included 33 respondents from SPLC. This is only a rather small subset of the overall community and we thus plan to further extend the survey.

## 6 CONCLUSIONS AND FUTURE WORK

We presented the results of a questionnaire-based survey we conducted with 33 SPLC 2018 participants to analyze their perceptions regarding SPL research. Quite a few topics are considered by survey respondents as needing further work, which also have already received a very significant amount of attention in the literature raising the question how to better preserve and communicate (SPL) knowledge. Also, while an analysis of survey responses and a comparison of survey responses with an earlier literature study reveal no significant differences between topics investigated by academia and industry, survey participants have a different view.

In our future work we plan to (i) conduct further surveys with more people, also outside of SPLC and (ii) perform in-depth interviews with academics and industry people to discuss our results in more detail and to uncover root causes.

## ACKNOWLEDGMENTS

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Primetals Technologies is gratefully acknowledged (Rabiser). This work was supported, in part, also by the ITEA3 project REVaMP2, funded by the BMBF (German Ministry of Research and Education) under grant 01IS16042H (Schmid), and by SFI grant 13/RC/2094 (Botterweck). Any opinions expressed herein are solely by the authors and not by the supporting agencies.

## REFERENCES

- [1] Mathieu Acher, Roberto E. Lopez-Herrejon, and Rick Rabiser. 2017. Teaching Software Product Lines: A Snapshot of Current Practices and Challenges. *TOCE* 18, 1 (2017), 2:1–2:31.
- [2] Vander Alves, Nan Niu, Carina Alves, and George Valen  a. 2010. Requirements engineering for software product lines: A systematic literature review. *Information and Software Technology* 52, 8 (2010), 806–820.
- [3] Sven Apel, Don Batory, Christian K  stner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.
- [4] Rabih Bashroush, Muhammad Garba, Rick Rabiser, Iris Groher, and Goetz Botterweck. 2017. CASE Tool Support for Variability Management in Software Product Lines. *ACM Comput. Surv.* 50, 1 (2017), 14:1–14:45.
- [5] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. 1999. PuLSE: A Methodology to Develop Software Product Lines. In *Proc. Symp. Softw. Reusability*. ACM, 122–131.
- [6] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej W  owski. 2013. A survey of variability modeling in industrial practice. In *Proc. 7th Int'l Workshop on Variability Modelling of Software-intensive Systems*. ACM, 7:1–7:8.
- [7] Stefan Biffl, Marcos Kalinowski, Rick Rabiser, Fajar Ekaputra, and Dietmar Winkler. 2014. Systematic Knowledge Engineering: Building Bodies of Knowledge from Published Research. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)* 24, 10 (2014), 1533–1571.
- [8] Lionel C. Briand, Domenico Bianculli, Shiva Nejati, Fabrizio Pastore, and Mehrdad Sabetzadeh. 2017. The Case for Context-Driven Software Engineering Research: Generalizability Is Overrated. *IEEE Software* 34, 5 (2017), 72–75.
- [9] Lianping Chen and Muhammad Ali Babar. 2011. A systematic review of evaluation of variability management approaches in software product lines. *Information & Software Technology* 53, 4 (2011), 344–362.
- [10] Marcus Ciolkowski, Oliver Laitenberger, Sira Vegas, and Stefan Biffl. 2003. Practical Experiences in the Design and Conduct of Surveys in Empirical Software Engineering. In *Empirical Methods and Studies in Software Engineering*, R. Conradi and A. Wang (Eds.). Springer, 104–128.
- [11] Krzysztof Czarnecki, Paul Gr  nbacher, Rick Rabiser, Klaus Schmid, and Andrzej W  owski. 2012. Cool features and tough decisions: a comparison of variability modeling approaches. In *Proc. 6th Int'l Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 173–182.
- [12] Jessica D  az, Jennifer P  rez, Pedro P Alarc  n, and Juan Garbajosa. 2011. Agile product line engineering—a systematic literature review. *Software: Practice and Experience* 41, 8 (2011), 921–941.
- [13] Emelie Engstr  m and Per Runeson. 2011. Software product line testing—a systematic mapping study. *Information and Software Technology* 53, 1 (2011), 2–13.
- [14] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. 2014. Variability in Software Systems - A Systematic Literature Review. *IEEE Trans. Software Eng.* 40, 3 (2014), 282–306.
- [15] Ruben Heradio, Hector Perez-Morago, David Fern  ndez-Amor  s, Francisco Javier Cabrerizo, and Enrique Herrera-Viedma. 2016. A bibliometric analysis of 20 years of research on software product lines. *Information and Software Technology* 72 (2016), 1–15.
- [16] Gerald Holl, Paul Gr  nbacher, and Rick Rabiser. 2012. A systematic review and an expert survey on capabilities supporting multi product lines. *Information & Software Technology* 54, 8 (2012), 828–852.
- [17] C Marimuthu and K Chandrasekaran. 2017. Systematic Studies in Software Product Lines: A Tertiary Study. In *Proc. 21st Int'l Systems and Software Product Line Conf.* ACM, 143–152.
- [18] Maira Marques, Jocelyn Simmonds, Pedro O. Rossel, and Maria Cecilia Bastarrica. 2019. Software product line evolution: A systematic literature review. *Information and Software Technology* 105 (2019), 190–208.
- [19] Jabier Martinez, Wesley K. G. Assun  o, and Tewfik Ziadi. 2017. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *Proc. 21st Int'l Systems and Software Product Line Conf.* ACM, 38–41.
- [20] Kannan Mohan, Balasubramanian Ramesh, and Vijayan Sugumaran. 2010. Integrating software product line engineering and agile development. *IEEE Software* 27, 3 (2010), 48–55.
- [21] Muhammad A Noor, Rick Rabiser, and Paul Gr  nbacher. 2008. Agile product line planning: A collaborative approach and a case study. *Journal of Systems and Software* 81, 6 (2008), 868–882.
- [22] Linda M Northrop and Lawrence G Jones. 2008. Introduction to software product line adoption. In *Proc. of the 12th International Software Product Line Conference*. IEEE, 371–372.
- [23] Klaus Pohl, G  nter B  ckle, and Frank van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer.
- [24] Mikko Raatikainen, Juha Tiihonen, and Tomi M  nnist  . 2019. Software product lines and variability modeling: A tertiary study. *Journal of Systems and Software* 149 (2019), 485–510.
- [25] Rick Rabiser, Paul Gr  nbacher, and Deepak Dhungana. 2010. Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Information & Software Technology* 52, 3 (2010), 324–346.
- [26] Rick Rabiser, Klaus Schmid, Martin Becker, Goetz Botterweck, Matthias Galster, Iris Groher, and Danny Weyns. 2018. A study and comparison of industrial vs. academic software product line research published at SPLC. In *Proc. of the 22nd Int'l Systems and Software Product Line Conf.* ACM, 14–24.
- [27] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villega. 2012. Software diversity: state of the art and perspectives. *STTT* 14, 5 (2012), 477–495.
- [28] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *Proc. 14th IEEE Int'l Conf. on Requirements Engineering*. IEEE, 136–145.
- [29] Frank van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software product lines in action - the best industrial practice in product line engineering*. Springer.
- [30] W Paul Vogt and R Burke Johnson. 2005. *Dictionary of Statistics and Methodology - A Non-technical Guide for the Social Sciences*. Sage Publications.

# Feature Oriented Refinement from Requirements to System Decomposition: Quantitative and Accountable Approach

Masaki Asano

Yoichi Nishiura

asano\_m@elec.aisin.co.jp

nishiura@elec.aisin.co.jp

Aisin Seiki Co., Ltd.

Kariya, Aichi, Japan

Tsuneo Nakanishi

tun@fukuoka-u.ac.jp

Fukuoka University

Fukuoka, Fukuoka, Japan

Keiichi Fujiwara

Fujiwara.Keiichi@mss.co.jp

Mitsubishi Space Software Co., Ltd.

Amagasaki, Hyogo, Japan

## ABSTRACT

This paper presents the revised domain engineering process to develop product lines of automotive body parts in Aisin Seiki Co., Ltd. In the process, feature analysis is conducted by a limited number of engineers with talent of abstraction and separation and other work including specifications and architecture design is conducted by average engineers who know the products. Feature analysis defines a hierarchy of abstraction, achieves separation of concerns, and disciplines other artifacts to follow the structure of abstraction and separation. Requirements and specifications are refined by the use case, use case scenario, and hierarchical tabular description (USDM) in a step-wise manner. The specification in USDM is refined to a system decomposition in a quantitative and accountable manner using the robustness diagram and design structure matrix. The revised domain engineering process reduced the issues pointed out in software reviews concerning errors on specifications and architecture design. Moreover, it reduced lead time for architecture design and produced the architecture tolerant to changes.

## CCS CONCEPTS

- Software and its engineering → Software product lines.

## KEYWORDS

software product lines, automotive body parts, feature analysis, use case approach, robustness analysis, design structure matrix

### ACM Reference Format:

Masaki Asano, Yoichi Nishiura, Tsuneo Nakanishi, and Keiichi Fujiwara. 2019. Feature Oriented Refinement from Requirements to System Decomposition: Quantitative and Accountable Approach. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19, September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3336294.3336314>

## 1 INTRODUCTION

Aisin Seiki Co., Ltd. is an automotive component manufacturing company founded in 1965, which belongs to the Toyota group. The core business of the company is manufacturing of automotive body parts. The company has designed, manufactured, and supplied automotive body parts to automobile manufacturers globally. It has recently strengthened sales of automotive body parts in the European market and also in the market of developing countries. That has been increasing the variety of parts in number ever since. On the other hand, competition for engineers in the Chukyo metropolitan area where the company is located has been fierce, and consequently, the company's developmental resources have remained the same. This circumstance motivated the company to migrate to product line development of the automotive body parts.

As reported in [1], the company has succeeded in migration to product line development of some automotive body part families. In the automotive domain, many successful case studies by various approaches have been reported so far[2–6]. The company adopts feature-oriented product line engineering that many successful adoptions has been reported in industries[7–13]; that is, it analyzes the system specifications of more than one variant of the existing product family and represents commonality and variability among the variants as a feature model[14]. Through construction and refinement of the feature model, it aligns the boundaries of the features based on the abstraction levels and concerns over them and organizes the relations among the features. It concurrently conducts feature analysis and structured analysis/design and establishes a product line architecture having the component boundaries coherent with the feature boundaries. This approach has remarkably reduced complexity of the software structure of the automotive body parts. The company developed a new variant in the so-called *clone-and-own* fashion before adopting the paradigm of software product lines. Repeated clone-and-owns had made the software structure and behavior incomprehensible. Clone-and-own derivation of a new variant required substantial cost to analyze the impact imposed by the changes in the software of the existent variant. However, as a result of migration to product line development, we have achieved remarkable cost reduction by prescribed derivation of the variant which eliminates the cost for the impact analysis. This successful case has heightened the interest in product line development within the company. The management has started to request the development support department in charge of domain engineering to disseminate the product line approach across the board.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336314>

The development support department has been implementing the approach to other automotive body parts and migrating their development to product line development, and has thereby improved the quality, cost, and the development work period. However, company's established approach requires one to two years for domain engineering. Automotive body parts have long life cycles of about three to five years, and once software has been created for certain automotive body parts, any major changes cannot be added easily to the software until the next life cycle starts. For this reason, the long development period of domain engineering can miss opportunities to migrate to product line development and make productivity of software development for the entire company stay low.

Aisin Seiki has recently committed itself to improving the domain engineering process established in the company to accelerate adoption of the software product line development approach in the entire company. It is essential for the acceleration to reduce the term for the established domain engineering process that has taken too much. In Section 2, the authors analyze possible reasons of protraction of the established domain engineering process based on experiences in previous projects and mention the solutions committed in the revised domain engineering process. The solutions are feature oriented use case modeling combining with hierarchical tabular description of requirements and specifications in pair, which is described in Section 3, and quantitative and accountable system decomposition for software architecture design by robustness analysis with the design structure matrix, which is described in Section 4. Section 5 evaluates the domain engineering process improved by these techniques. Section 6 describes related work. Finally, Section 7 concludes the paper.

## 2 PROBLEMS AND SOLUTIONS

### 2.1 Reasons of Protracted Domain Engineering

To quickly deploy product line development to the entire company, it is necessary to reduce the time required for domain engineering for each product family. The authors have identified possible reasons of protracted domain engineering after analyzing the past experience of domain engineering activities conducted by the company.

*Different recognition of sophisticated software architecture among engineers:* The company has engineers with various experiences and skills. They have different recognition of software architecture. The abstraction levels for development document descriptions and software structures that they consider as optimal are not consistent. The development support department in charge of domain engineering asks them to provide support in feature analysis and structured analysis; however, their artifacts often deviate from expectation of the development support department especially in abstraction and separation. More engineers are needed to speed up domain engineering, but more rework for analysis can happen as the number of the engineers involved increases.

*Shortage of engineers who can conduct feature analysis:* It is essential to have a comprehensive view on a product family, define a hierarchy of abstraction and achieve separation of concerns for the specifications of the product family, and construct an architecture durable to changes for successful product line development. Feature

analysis plays an important role to discipline this domain engineering process. However, not all of the engineers are skilled at feature analysis. Many engineers in the company are well-experienced in considering what specific means can realize given requirements from end to end, but as they have performed derivative development processes in the clone-and-own fashion for a long time, they have had little experience in getting a macroscopic view of different variants, abstracting domain technologies, and representing them as features.

*Concurrent conduction of feature analysis and structured analysis/design:* The domain engineering process established in the company[1] requires concurrent conduction of feature analysis and structured analysis/design as shown in Figure 1. The feature analysis is conducted to define a hierarchy of abstraction and achieve separation of concerns, namely for pre-design of software architecture. The structured analysis/design is conducted to define a software architecture for the product line. The artifacts of the both analyses, namely the feature model and data flow diagram (DFD), are required to be consistent; however, there is a considerable semantic gap between them. The feature model defines abstract concepts in different abstraction levels with variability constraints, while the DFD represents data transformation to realize functional requirements. This semantic gap makes it difficult to divide the process, and consequently, a heavy burden is imposed on a limited number of developers who are skilled at feature analysis.

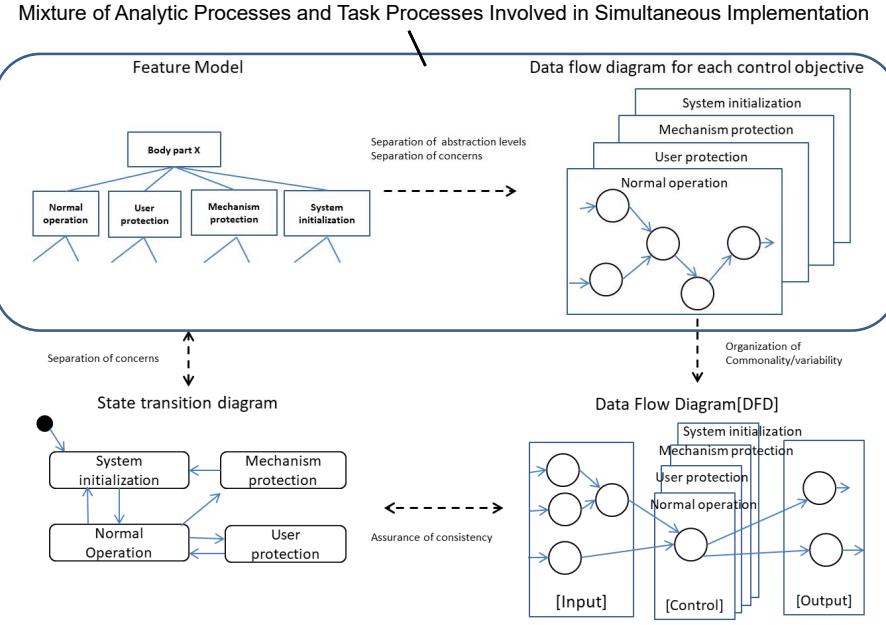
*Quality issues of automotive body parts software:* Process based development such as Automotive SPICE[15] is popularly conducted in the automobile industry. It must be logically explained that the artifacts created through the process are consistent and the designs completely fulfill requirement specifications. The domestic design documents are manually authored in the natural language with semi-formal representations such as UML. Consistency among these artifacts is guaranteed basically by engineers' review. Therefore, bigger and more complicated products require more manpower for the review.

Moreover, the domain engineering process established in the company relies on the experience and skills of the individual engineers to conduct structured analysis/design. That makes it difficult to verify whether every requirement specification is fulfilled as data flows in the DFD in a uniform manner and increases manpower needed for verification tasks.

### 2.2 Solutions

In light of the above-mentioned possible reasons of protracted domain engineering, the authors improved the established domain engineering process of the company to shorten time required for domain engineering and to accelerate adoption of the software product line paradigm in development of more product families in a shorter time as below.

The authors defined a requirement and specification sub process by use case approach in the revised domain engineering process, which achieved separation of feature analysis and architecture design. Only a limited number of talented engineers with good sense of abstraction can conduct feature analysis. On the other hand, once a hierarchy of abstraction and separation of concerns are established as a result of feature analysis, many average engineers can



**Figure 1: Company's established domain engineering process**

conduct architecture design at better quality. This separation of feature analysis and architecture design releases talented engineers from overload due to concurrent conduction of the both activities, avoids serialization of domain engineering work for multiple product families due to resource limitation, and facilitates parallel migration of multiple product families to product line development.

In the revised domain engineering process, requirements are modeled as use cases and then refined as use case scenarios. They are further refined as specifications through hierarchical tabular description in the USDM (Universal Specification Describing Manner)[16, 17] as the authors mention in the next section. The refinement process is guided based on the abstraction hierarchy and separation of concerns given by the feature model and able to be conducted by many average engineers. In the USDM, requirements and specifications satisfying them are described in pairs; thus, it can be examined easily whether specifications are defined to satisfy all the requirements completely.

Moreover, the authors defined an architecture design process in a quantitative and accountable manner with robustness analysis of the ICONIX process[18]. The robustness analysis is a relatively well-defined process which transforms requirements described in use case scenarios into a system decomposition. In the company's established domain engineering process, system decomposition was conducted according to engineer's individual skills and experience. In the revised domain engineering process, system decomposition is performed by the robustness analysis in a quantitative manner with using the design structure matrix (DSM)[19]. Since the architecture design process transforms the specifications well-structured in the USDM to a system decomposition in a well-defined manner, it is easy to verify whether the architecture satisfies the functional specifications completely. Moreover, engineers can consent to system decomposition, if it is given with quantitative reasons.

Figure 2 shows the revised domain engineering process.

### 3 FEATURE ORIENTED USE CASE MODELING

The purpose of use case analysis is to capture the functional requirements of a given system by describing how the system interacts with its stakeholders called as *actors*. Use case analysis allows us to embody the functional requirements of a system, which are abstract and elusive at the beginning of development, as a flow of information, events, and responses exchanged between the system and its actor(s). This facilitates sharing of the vision related to the system functional requirements among the stakeholders of the development. On the other hand, it requires skill to define use cases and describe their scenarios, interaction between the system and its actors, in reasonable granularity and abstraction level coherently with variability among the products. Further, in the case of a system such as the automotive body system, in which multiple functions access the same sensors and actuators, use case scenarios may include substantial duplication.

For this reason, in the revised domain engineering process of the company, we first conduct feature analysis to identify the concepts and functions regarding a given system as its features, organize the abstraction levels among them in a hierarchical form, and separate the concerns. At the same time, we find partial semantic overlap among the features as sub features shared by multiple features.

#### 3.1 Abstraction and Separation of Concerns by Feature Analysis

Feature analysis is conducted to identify features representing requirements, specifications, and technologies used for implementation from the system specification of existing products and organize them in a hierarchical form to define a hierarchy of abstraction

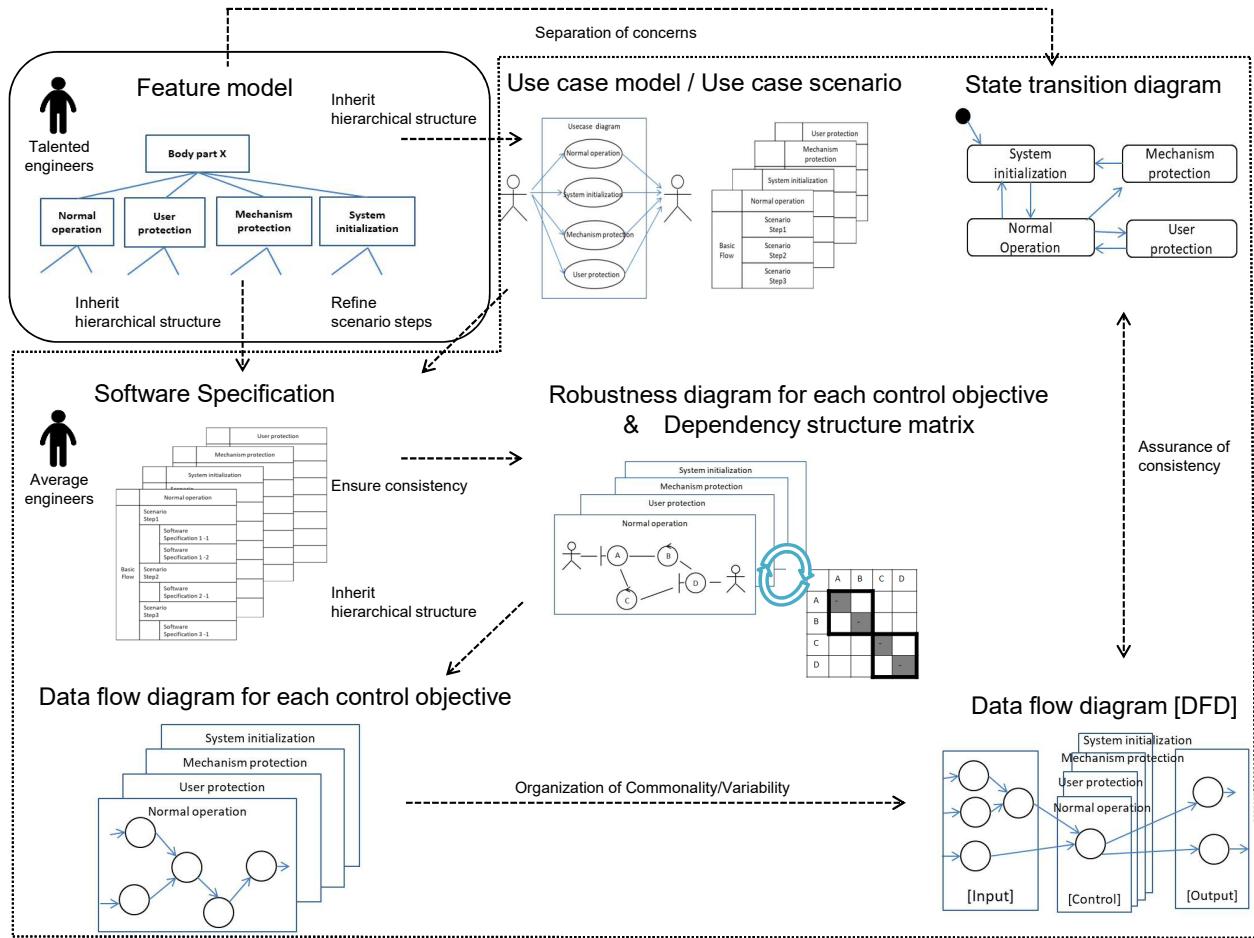


Figure 2: Company's revised domain engineering process

and achieve separation of concerns. The feature model produced as a result of analysis must include features representing requirements and also features representing specifications that satisfy the requirements. A feature of requirements should be complemented if it is not defined with features of specifications realizing the requirements — and *vice versa*. Only a limited number of engineers have excellent sense of abstraction and separation. Feature analysis should be performed by such talented engineers. However, we should not charge them with other domain engineering activities that average engineers can do. The other domain engineering activities in the process proposed in this paper are designed to be disciplined by the result of feature analysis. The primary objective that we conduct feature analysis first is to separate activities that only talented engineers can do from ones that average engineers can do, optimize human resource allocation, and achieve efficient division of work.

As the authors described in [1], in Aisin Seiki, system specifications tended to be excessively refined to the abstraction level of technical concrete solutions without describing requirements that are objectives of the specifications. Therefore, engineers responsible for feature analysis in the company are often required to

consider why the system specifications are needed and find features representing requirements.

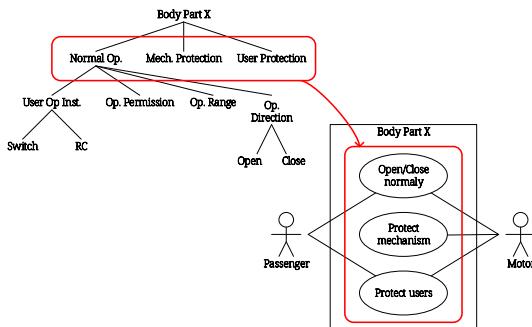
### 3.2 System Requirement Description with Use Cases

As a result of feature analysis, functional requirements of the system and specifications realizing the requirements are organized hierarchically. Their abstraction levels are separated in the hierarchy and their concerns are also separated. Moreover, partial semantic duplication of them are identified. Next, we define use cases, describe their use case scenarios, and further refine them into specifications with reasonable granularity and abstraction levels in a coherent form with the structure of the feature model. Duplicated or variable parts of specifications are extracted with using mechanisms such as generalization, extension, and/or inclusion of use cases. The extending or included use cases also are extracted with reasonable granularity and abstraction levels in a coherent form with the structure of the feature model.

The process for use case modeling proposed in the revised domain engineering process of the company is as follows.

First, we identify services modeled as top-level use cases corresponding to features in upper layers of the feature model. The feature model consists of features with higher abstraction levels representing objectives of the system functions in the higher layers and those with lower abstraction levels representing means of the system functions in the lower layers. For this reason, we select a group of layers in which the service level features are arranged and define the use cases based on the features in those layers. The service is a process that is triggered from its actor or system itself, performs a certain judgement and computation, and returns a response to an actor and also that is recognized as a coarse grain functional capability by system stakeholders. In Figure 3, the features immediately under the root of the feature model correspond to the use cases in the use case model in a one-to-one correspondence.

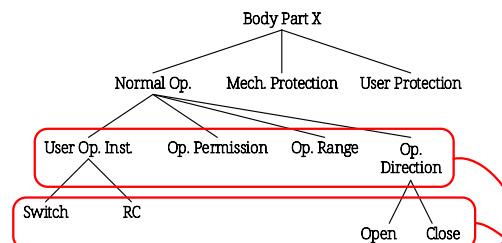
Service level features can appear in multiple layers. If a service level feature in the lower layer has fine granularity, we regard the feature as just a small part of the use case realizing its parent service feature in the upper layer; *i. e.* we regard the both features as ones corresponding to the same use case. However, if a service level feature in the lower layer has coarse granularity, we regard the feature as a single use case depending on the use case realizing its parent service feature in the upper layer; *i. e.* we regard the feature as one corresponding to the extending or included use case and the parent feature as one corresponding to the base use case.



**Figure 3: Correspondence between feature model and use case diagram**

Second, we describe use case scenarios for the use cases in the constructed use case model. The abstract use case scenario and concrete use case scenario are described in parallel for each use case by average engineers. Intent of each interaction between the system and actor is described in the abstract use case, while action relating to the interaction is described in the concrete use case scenario. As shown in the example of Figure 4, a use case corresponds to a service level feature (in the second layer in the example); the abstract use case scenario of the use case corresponds to sub features of the service level feature (in the third layer in the example); and the concrete use case scenario of the use case corresponds to sub sub features of the service level features (in the fourth layer in the example). The features in much lower layers (in the fifth layer or lower in the example) are in the abstraction levels lower than that of use case scenario; thus, they never appear in use case scenarios.

Generally, it is regarded as inappropriate to include concrete actions relating to the interaction between the system and actors such as user interface affairs in the use case scenario. We are often forced to change use case scenarios with concrete actions by changes of user's preference on user interface even in the case requirements themselves are not changed. Abstract use case scenarios are tolerant to changes, thus reusable. However, in many cases, engineers unfamiliar with software engineering from control engineering, mechanical engineering, etc. prefer concrete use case scenarios when they discuss the system under consideration with software engineers. For this reason, the process proposed in this paper dares to be designed to describe both abstract and concrete use case scenarios in parallel to make them be a common ground among all the engineers to comprehend requirements of the system.



**Figure 4: Correspondence between feature model and use case scenarios**

Finally, we realize variability represented by the feature model as variation points in the use case model and/or use case scenarios. Coarse grain variability (or use case level variability) is realized by extension and/or inclusion of the use case, while fine grain variability (or use case scenario level variability) is realized by use case variation points embedded in the use case scenario. If and only if a feature is selected on product derivation and also on execution, the extending/included use or use case variation point relating to the feature is activated.

### 3.3 Software Specification Description by Refining Use Case Scenarios

After we describe requirements of the system as the use case model and use case scenarios, we describe the software requirement specifications. The software requirement specifications are description what the software should do to realize the requirements without any ambiguity.

We employ the USDM (Universal Specification Describing Manner) to describe the software specifications with extension representing variability for product line development. The USDM[16, 17], which is widely adopted by the Japanese industry, is a hierarchical tabular description of software requirements and specifications. The USDM clearly separates the requirements and specifications. The requirements are allowed to be described with some ambiguity in general, while the specification must not have any ambiguity since they are handed over to the design process. The USDM requires us to describe each requirement in a row and a group of specifications realizing the requirement just below the row with reasons why the requirement and specifications are needed. Since there can be a hierarchy based semantic relationships such as generalization/specialization, composition, *etc.* among requirements in general, it is allowed that a row of a requirement has nested rows of its sub requirements. It is easily examined whether requirements can be fully satisfied by specifications in a traceable manner thanks to paired description of requirements and specifications and also hierarchical description of requirements with their sub requirements. Figure 5 shows an example of partial description in the USDM, which describes requirements and specifications of an imaginary automotive body part that open and close a mechanism. The outermost row in the table describes a top-level requirement of the system. Descriptions of the sub requirements composing the requirement are nested in the row. Decomposition of the requirement into its sub requirements is conducted recursively; thus, the nesting structure of the table becomes similar to the recursive decomposition structure of the requirements. Finally, specifications satisfying the (sub) requirement is described in the row of the (sub) requirement.

It requires engineers of sense of abstraction and separation to describe software requirements and specifications at appropriate abstraction levels for each nesting level in the USDM, as construction of use case models and description of use case scenarios do so. In fact, many projects in the company hesitate to introduce the USDM for this reason, although they understand benefits of describing software requirements and specifications in the USDM. The revised domain engineering process of the company is designed so that we comply the hierarchy of the abstraction given by the feature model to refine software requirements written in the use case scenarios into software specifications in the USDM. The refinement is conducted as follows.

First, we describe steps in a use case scenario in a hierarchical manner in the USDM. We describe the steps of the abstract use case scenario as the first level requirements in the USDM (such as R-01 in Figure 5 for example). For each first level requirement in the USDM, we describe the corresponding steps of the concrete use case scenario as the second level requirements in the USDM (such as R-01-01, R-01-02, ..., in Figure 5 for example). If any steps of the

concrete use case scenario are not present, we do not describe the second level requirements.

Second, we describe software specifications in the USDM based on the abstraction hierarchy given by the feature model. In concrete, we describe software specifications corresponding to the features that do not appear neither in the use case model nor in the use case scenarios (features in the fourth layer or lower in the example shown in Figure 6) as specifications described in the third layer or lower of the USDM (such as S-01-01-1 in Figure 5). The company imports the specifications described in the USDM from the existing system specifications with reforms, since the company is migrating to product line development.

Third, we describe commonality and variability of software specifications in the USDM. So far, we have described software requirements and specifications in the USDM in coherent with the feature model. Therefore, each variable software specification can be included or excluded depending on the feature selection. We annotate each specification by either common or variable attributes, not by the feature that it realizes, since traceability from the feature to the specification is maintained in a concentrated manner by the traceability matrix, not by the USDM.

## 4 ESTABLISHING A SOFTWARE ARCHITECTURE WITH ROBUSTNESS ANALYSIS

As mentioned in Section 2, the established domain engineering process of the company[1] tends to be protracted due to its way of software architecture design that transforms requirements/specifications into a system decomposition without any common guidelines. The company considered to introduce robustness analysis of the ICONIX process[18] which is widely known as a powerful method refining requirements described as use case scenarios into a system decomposition. However, robustness analysis also depends on engineer's individual sense, when it assign controls representing *atomic* behavioral requirements to components. Mitsubishi Space Software (MSS), a company constructing special software used for space systems *etc.* to which one of the co-authors of this paper belongs, has practiced a method to decompose the system and assign behavioral requirements/specifications described in the USDM to components in a quantitative and accountable manner that uses the design structure matrix (DSM)[19] in robustness analysis. Aisin Seiki integrated the method for system decomposition into its revised domain engineering process to design a software architecture.

### 4.1 Quantitative and Accountable System Decomposition

The robustness analysis extended by MSS constructs a robustness diagram from the software specifications in the USDM. The robustness diagram consists of the following three types of objects:

- *boundary* representing interface between the system and its outside
- *control* representing a certain activity of the system
- *entity* representing data handled in the system

UC01	Operate the body part X normally		
Reason	The electric opening and closing movement of the body part X must be operated according to the intention of the passenger.		
Primary Actor	Passenger, Body Part X		
Requirements	R-01	A request for operation from the passenger is sent to the system and interpreted as an activation event.	
		Reason	A request for operation from the passenger is required to operate the body part X electrically.
			The system determines that an operation from the operation switch is the activation event.
		Reason	Because the operational input by the passenger to open the body part X is input to the system as an operational event from the operation switch.
		Spec.	Determine an activation event: According to the operational event from the operation switch, the system determines whether the event is an activation event for the opening direction listed below: (1) Activation event for the opening direction with the manual operation. - Condition A - Condition B - Condition C (2) Activation event for the opening direction with the automatic operation
	R-01-02	****	
		Reason	***

Figure 5: USDM Example

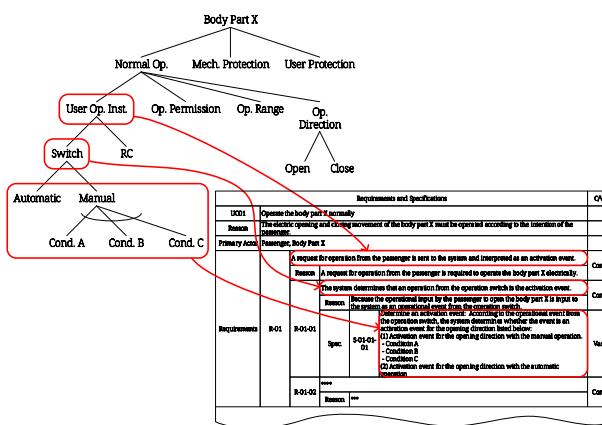


Figure 6: Correspondence between feature model and software requirement specification

To construct the robustness diagram from the software specifications in USDM, i) we define a control for each specification described in the USDM; ii) we define a boundary for each actor in the use case model; iii) we define entities for data mentioned in the software specifications; iv) we connect the controls, boundaries, and entities based on the software specification. We conduct the

above-mentioned process i) to iv) for all the use cases and aggregate all the results into one robustness diagram.

The MSS's method fuses controls of the robustness diagram with using DSM as mentioned below to find a system decomposition with components of high cohesion and low coupling in a quantitative and accountable manner.

First, we construct a design structure matrix (DSM) that represents coupling strengths among controls in the robustness diagram. Rows and columns of the matrix have one-to-one correspondence with controls of the robustness diagram; that is,  $i$ -th row and  $i$ -th column correspond to the control  $i$ . We assign the coupling strength between  $i$ -th and  $j$ -th controls to the  $(i, j)$  element of the matrix. Here, we define the coupling strength between any couple of controls  $c_a$  and  $c_b$  ( $c_a \neq c_b$ ) in the robustness diagram, denoted by  $c(c_a, c_b)$ , as follows:

$$c(c_a, c_b) = \begin{cases} \sum_{p \in P(c_a, c_b)} w(p) & \text{(if } P(c_a, c_b) \neq \emptyset) \\ 0 & \text{(if } P(c_a, c_b) = \emptyset) \end{cases}$$

where  $P(c_a, c_b)$  is the set of the paths between  $c_a$  and  $c_b$  that includes zero or one boundary or entity and  $w(p)$  is a weight of any path in  $p \in P(c_a, c_b)$ .  $w(p)$  is defined as one (1) if  $c_a$  and  $c_b$  are directly connected or as two (2) if there is one boundary or entity in  $p$ .

Next, we apply a sort of block diagonalization to the constructed DSM; that is, we collect a bunch of bigger numbers close to the

diagonal blocks of the matrix by permuting the controls. Note that permutation of the controls is equivalent to identical permutation of rows and columns of the DSM. The block diagonalization is never mathematically-rigid. It is allowed a small amount of non-zero elements remain in non-diagonal blocks of the matrix and also a small amount of zero elements remain in diagonal blocks. The block diagonalization is performed manually with help of color visualization of the matrix. The controls corresponding to a diagonal block have strong coupling each other; therefore, the functions of the controls should be assigned to the same component, namely a process in the data flow diagram constructed by structured analysis/design.

Figure 7 shows the robustness diagram. We construct a DSM and apply block diagonalization to the DSM by the above-mentioned process and, as a result, the DSM is block diagonalized as shown in Figure 8. The diagonal blocks A to E found in the DSM of Figure 8 correspond to the partitions A to E represented in the robustness diagram of Figure 7. The functions of the controls in each partition are assigned to a single component.

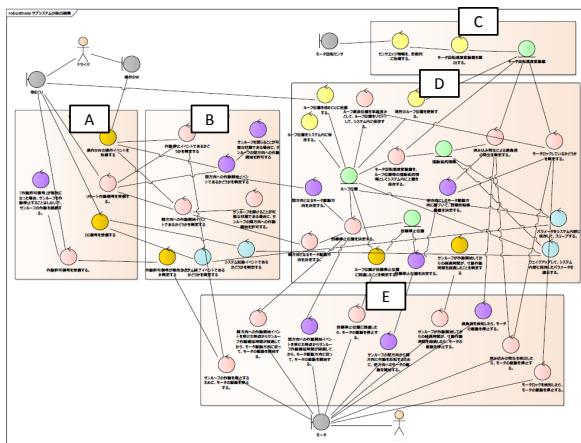


Figure 7: Robustness diagram before improvement

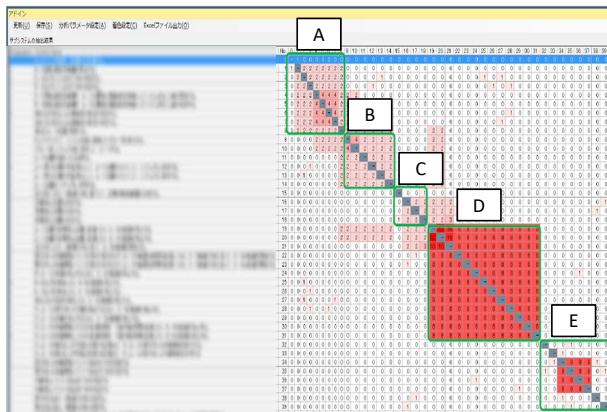


Figure 8: DSM created from robustness diagram before improvement

## 4.2 Extension of MSS's System Decomposition Process

MSS's process for quantitative and accountable system decomposition sometimes assign many controls to a single component such as component D in Figure 7. That leads components with insufficient separation of roles. The boundary and control of the robustness diagram are reduced from an actor of the use case model and a software specification in the USDM, respectively. However, the entity of the robustness diagram can be defined by engineer's individual sense without any rules or guidelines. Therefore, the process can produce components with insufficient separation of roles, since the coupling strength between controls among an entity gets excessively bigger depending on definition of entities.

To resolve the problem and also to produce components subject to the abstraction hierarchy and separation of concerns given by the feature model, Aisin Seiki established an additional rule for definition of entities on adoption of MSS's system decomposition process. The rule is that we model only the data handled in the topmost specification as entity.

Figure 10 shows a result of block diagonalization by MSS's system decomposition process with Aisin Seiki's additional rule of entity definition. Figure 9 shows partitions of the controls defined by the result of block diagonalization. Note that there is no bigger partitions in the robustness diagram; that is, functions of controls are well distributed to components.

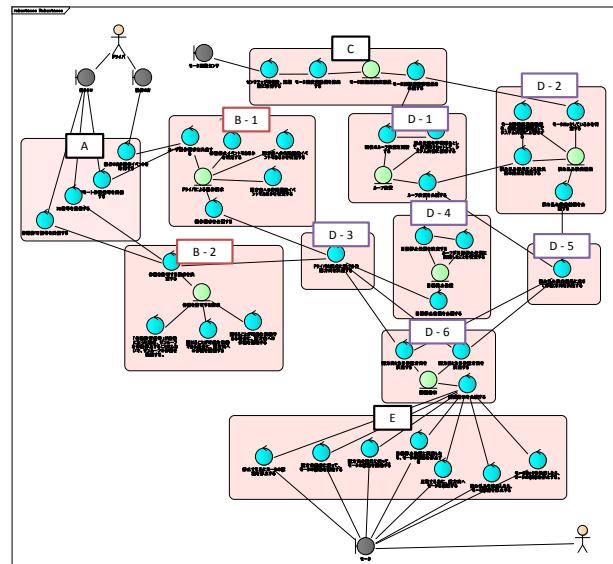
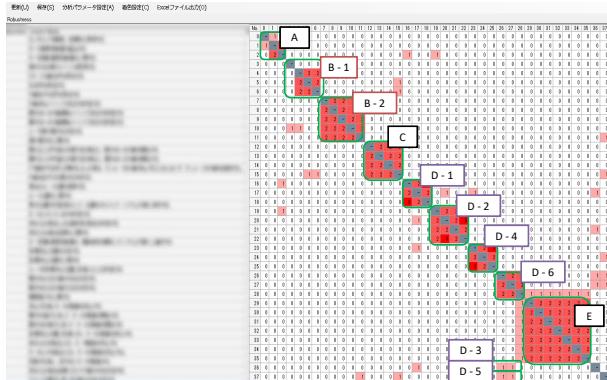


Figure 9: Robustness diagram after improvement

MSS's process does not consider product line development. Automotive body parts are basically control intensive. Software for automotive body parts handle events and responses to them mainly and only a limited kinds of data. Functions themselves provided by automotive body parts are not so different among product variants. Since variability tends to appear in conditions relating to behavioral changes of the mechanism, most of variability can be enclosed in a single component. Therefore, MSS's process works enough even if it



**Figure 10: DSM created from robustness diagram after improvement**

does consider boundaries of features for product line development of automotive body parts. Moreover, variability is well separated and localized within a limited scope of specifications thanks to step-wise refinement of requirements and specifications subject to the results of feature analysis. That also contributes to make MSS's process work.

## 5 EVALUATION

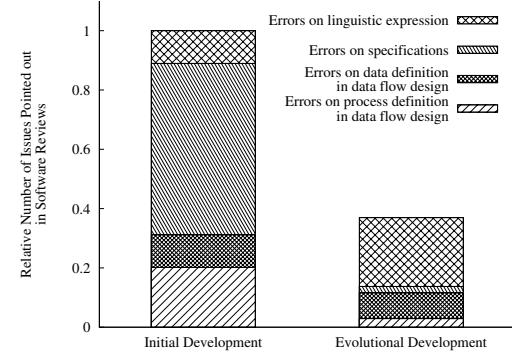
Aisin Seiki migrated to product line development to develop a family of an automotive body part X by following the domain engineering process[1]. In this initial development, feature analysis was conducted by an engineer with talent of abstraction. Initially, architecture design was conducted by another average engineer with knowledge on the product to reduce the development time. However, due to not a little amount of rework for software architecture design, the engineer who conducted feature analysis ended up doing feature analysis and software architecture design concurrently.

The company evolved the established product line of the automotive body part X to add features that had not been intended on the initial development. The company improved the domain engineering process as described in this paper and applied it to the evolutional development of the product line. In this evolutional development, feature analysis was conducted by the engineer with talent of abstraction and architecture design was conducted by another engineer with knowledge on the product. The both activities were conducted in a completely separated manner as the revised domain engineering process.

### 5.1 Disciplined Specifications and Architecture Design

The authors examined the number of issues pointed out in software reviews for architecture design to validate whether average engineers could conduct architecture design by structured analysis and design in the revised domain engineering process as the talented engineer who conducted feature analysis intended. Figure 11 shows the results. Since the scale of the initial and the evolutional developments are different, the number of issues pointed out in software

reviews in each development is divided by the scale of the respective development; that is, these numbers are the number of issues pointed out in a unit size of the constructed software. Moreover, the numbers are normalized by the total number of issues pointed out in the initial development.



**Figure 11: Results of the software review for architecture design**

As the figure shows, the number of issues pointed out in software reviews in the evolutional development was less than two-fifths of that in the initial development. Software reviews in the initial development pointed out many issues relating to errors on specifications and process definition in the data flow design. On the other hand, in the evolutional development, these kinds of issues were drastically reduced. Most of issues pointed out were relating to minor errors on linguistic expression. However, issues relating to errors on data definition in the data flow design were reduced but still pointed out. Based on these results, it can be safely concluded that, with the revised domain engineering process, engineers can describe specifications in the USDM and conduct system decomposition to processes by structured analysis/design as the feature model disciplines, although it leaves room for improvement on disciplined data flow design.

### 5.2 Lead Time for Architecture Design

The lead time for architecture design, the term from the start of feature analysis to the end of structured analysis/design, was measured. Since the scales of the initial and the evolutional developments are different, the measured lead time in each development is divided by the scale of the respective development; that is, the lead time for a unit size of the software constructed in the evolutional development was about half as much as that in the initial development.

The reduction was achieved by complete separation of feature analysis and architecture design and definition of the process for quantitative and accountable system decomposition. These improvements in the domain engineering process contributed to reduce time used for software reviews and rework in architecture design.

### 5.3 Architecture

In the established domain engineering process of the company, engineers have decomposed the system based on their skills and experience. In the revised domain engineering process, engineers decompose the system based on the results of robustness analysis with quantitative evaluation using the design structure matrix. The authors compared how the software architectures constructed by these domain engineering processes are durable to changes.

The structural impact ratio[20] was used for the comparison. The structural impact ratio indicates the degree of impact range brought by changes to a module. The impact ratio  $IR$  is defined as follows:

$$IR = \frac{1}{SLoC} \sum_{m \in S} \frac{\text{LoC}(m)}{SLoC} \sum_{m' \in \text{dep}(m)} \text{LoC}(m'),$$

where  $S$  is a set of the modules,  $\text{dep}(m)$  is a set of the modules depending on a module  $m \in S$  including  $m$  itself,  $\text{LoC}(m)$  is the number of source code lines of a module  $m \in S$ , and  $SLoC$  is the number of the whole source code lines. The impact ratio is not simply defined as the average number of dependent modules but on the number of source code lines for more precision. Bigger modules can be changed at more possibility; thus, the coefficient  $\text{LoC}(m)/SLoC$  is multiplied to the number of source code lines in the definition of the ratio.

The impact ratio of the software produced in the evolutional development was 1.1 times as much as that of the software produced in the initial development was 11%. The revised domain engineering process could shorten the lead time in domain engineering while keeping the improvement effects on the structural aspect of the software.

## 6 RELATED WORK

Gomaa presents a use case modeling approach for product line engineering[21]. The approach conducts use case modeling and describes use case scenarios to identify common and variable interactions between the system and its actors before feature analysis. The approach models bigger variable interactions as entire use cases, extending use cases, or included use cases and smaller variable interaction as use case variation points representing modification to be applied to the use case scenarios. The use cases, extending or included use cases, or use case variation points for a certain behavior or characteristic of the system, namely a feature, are activated together. That is, the feature is a concept tagging requirements reused and activated together for a product variant. The feature is not used for abstraction or separation of concerns in the approach. Moreover, the approach can produce use case scenarios including entities of various abstraction levels, since it captures commonality and variability with different abstraction levels among product variants in the use case model. On the other hand, the approach presented in this paper uses the feature as the first class concept for abstraction and separation of concerns to discipline refinement of requirements and specifications as well as design of software architecture. It produces abstract and concrete use case scenarios to separate abstraction levels; moreover, the concrete use case scenarios are refined to specifications in the USDM.

Lee *et al.* proposes a feature oriented approach to object-oriented design of software architecture for the product line[22]. In their process, feature analysis is conducted to define a hierarchy of abstraction and achieve separation of concern and object identification leading to software architecture design is disciplined by the constructed feature model. They define guidelines how features are refined to objects depending on categories of features, which are layered based on roles and abstraction levels. The domain engineering process proposed in this paper learned from their approach, but focuses on step-wise refinement of requirements and specifications before construction of the object model and conducts system decomposition based on the refined specification with help of robustness analysis. Accountability on safety is strictly required in development of automotive systems. The proposed process is designed so that we can guarantee traceability of artifacts from requirements to components to fulfill the accountability.

## 7 CONCLUSION

In this paper, the authors presented the revised domain engineering process for product line development of automotive body parts in Aisin Seiki Co., Ltd. The established domain engineering process of the company[1] conducts feature analysis and structured analysis/design concurrently; thus, it was difficult to divide the work for domain engineering and a limited number of engineers with talent of abstraction and separation had to dedicate domain engineering for a single product family at a time. This circumstance had inhibited speedy adoption of product line engineering.

Therefore, in the revised domain engineering process, the authors separated feature analysis, which can be conducted well only by talented engineers, and other work including specifications and architecture design, which can be conducted by average engineers who know the products. Feature analysis defines a hierarchy of abstraction, achieves separation of concerns, and further, disciplines step-wise refinement of requirements and specifications by the use case, use case scenario, and USDM and architecture design by structured analysis/design. Robustness analysis is employed to fill the semantic gap between specifications and system decomposition. The robustness diagram is constructed by rule from specifications in the USDM and system decomposition is planned on the diagram in a quantitative and accountable manner with help of the dependence structure matrix. This rule based approach eases consensus on architecture design among relevant engineers and reduces rework of architecture design. The number of issues pointed out in software reviews relating to errors on specifications and architecture design were drastically reduced in the revised domain engineering process.

## REFERENCES

- [1] Y. Nishiura, M. Asano, and T. Nakanishi, "Migration to Software Product Line Development of Automotive Body Parts by Architectural Refinement with Feature Analysis," *Proc. 25th Asia-Pacific Software Engineering Conference (APSEC 2018)*, pp. 522–531, Dec. 2018.
- [2] S. Thiel and A. Heim, "Modeling and Using Product Line Variability in Automotive Systems," *IEEE Software*, Vol. 19, No. 4, pp. 66–72, July/Aug. 2002.
- [3] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber, "Introducing PLA at Bosch Gasoline Systems: Experiences and Practices," *Proc. 3rd Software Product Line Conf. (SPLC 2004)*, pp. 34–50, Aug. 2004.
- [4] R. Flores, C. Krueger, and P. Clements, "Mega-Scale Product Line Engineering at General Motors," *Proc. 16th Int. Software Product Line Conf. (SPLC 2012)*, Vol. II, pp. 259–268, Sep. 2012.

- [5] T. Iida, M. Matsubara, K. Yoshimura, H. Kojima, and K. Nishino, "PLE for Automotive Braking System with Management of Impacts from Equipment Interactions," *Proc. 20th Int. Systems and Software Product Line Conf. (SPLC2016)*, pp. 232–241, Sep. 2016.
- [6] K. Hayashi, M. Aoyama, and K. Kobata, "Agile Tames Product Line Variability: An Agile Development Method for Multiple Product Lines of Automotive Software Systems," *Proc. 21st Int. Systems and Software Product Line Conf. (SPLC 2017)*, pp. 180–189, Sep. 2017.
- [7] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architecture," *Annals of Software Engineering*, Vol. 5, No. 1, pp. 143–168, 1998.
- [8] K. C. Kang, S. Kim, J. Lee, and K. Lee, "Feature-Oriented Engineering of PBX Software for Adaptability and Reusability," *Software: Practice and Experience*, Vol. 29, No. 10, pp. 875–896, Oct. 1999.
- [9] K. C. Kang, M. Kim, J. Lee, and B. Kim, "Feature-Oriented Re-engineering of Legacy Systems into Product Line Assets: A Case Study," *Proc. 9th Int. Software Product Line Conf. (SPLC 2005)*, pp. 45–56, Sep. 2005.
- [10] K. C. Kang, J. J. Lee, B. Kim, M. Kim, C. W. Seo, and S. L. Yu, "Re-engineering a Credit Card Authorization System for Maintainability and Reusability of Components: A Case Study," *Proc. 9th Int. Conf. on Software Reuse (ICSR2006)*, pp. 156–159, June 2006.
- [11] T. Iwasaki, M. Uchiba, J. Ohtsuka, K. Hachiya, T. Nakanishi, K. Hisazumi, and A. Fukuda, "An Experience Report of Introducing Product Line Engineering across the Board," *Proc. 14th Int. Software Product Line Conf. (SPLC 2010)*, pp. 255–258, Sep. 2010.
- [12] J. Otsuka, K. Kawarabata, T. Iwasaki, M. Uchiba, T. Nakanishi, K. Hisazumi, and A. Fukuda, "Small Inexpensive Core Asset Construction for Large Gainful Product Line Development: Developing a Communication System Firmware Product Line," *Proc. Joint Workshop of the 3rd International Workshop on Model-Driven Approaches in Software Product Line Engineering and 3rd Workshop on Scalable Modeling Techniques for Software (MAPLE/SCALE2011)*, 5 pages, Aug. 2011.
- [13] L. P. Tizzei, M. Nery, V. C. V. B. Segura, and R. F. G. Cerqueira, "Using Microservice and Software Product Line Engineering to Support Reuse of Evolving Multi-Tenant SaaS," *Proc. 21st Int. Systems and Software Product Lines Conf. (SPLC 2017)*, pp. 205–214, Sep. 2017.
- [14] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Analysis," Technical Report, CMU/SEI-90-TR-21, SEI/CMU, Nov. 1990.
- [15] Automotive SPICE, <http://www.automotivespice.com/>.
- [16] Y. Shimizu, *A Technique to Describe Requirements and Transform Them into Specifications* (Original Title: *Youkyu wo shiyouka suru gijutsu, hyougen suru gijutsu*), Gijutsu Hyohronsha, 2010. (in Japanese)
- [17] K. Kobata, E. Nakai, and T. Tsuda, "Process Improvement Using XDDP: Application of XDDP to the Car Navigation System," *Proc. 5th World Congress for Software Quality*, 8 pages, Nov. 2011.
- [18] D. Rosenberg and M. Stephens, *Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007.
- [19] T. R. Browning, "Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions," *IEEE Trans. on Engineering Management*, Vol. 48, NO. 3, pp. 292–306, Aug. 2001.
- [20] W. Okamoto, "Software Structure Diagnosis Method to Evaluate and Improve Design Maintainability," *Toshiba Review*, Vol. 68, No. 9, pp. 42–45, Sep. 2013. (in Japanese)
- [21] H. Gomaa, *Designing Software Product Lines with UML*, Addison-Wesley, 2005.
- [22] K. Lee, K. C. Kang, W. Chae, and B. W. Choi, "Feature-Based Approach to Object-Oriented Engineering of Applications for Reuse," *Software: Practice and Experience*, Vol. 30, No. 9, pp. 1025–1046, Sep. 2000.

# Enabling Automated Requirements Reuse and Configuration

Yan Li

State Key Laboratory of Software Development  
Environment, Beihang University  
Beijing, China  
yanll@buaa.edu.cn

Shaukat Ali

Simula Research Laboratory  
Oslo, Norway  
shaukat@simula.no

Tao Yue

Simula Research Laboratory, Oslo Norway and Nanjing  
University of Aeronautics and Astronautics, Nanjing,  
China  
tao@simula.no

Li Zhang

Beihang University  
Beijing, China  
lily@buaa.edu.cn

## ABSTRACT

Software-intensive systems belonging to a product line (PL) often have their shared architecture design available before they are developed. Therefore, the PL often has a large number of reusable and configurable requirements, which are naturally organized hierarchically based on the architecture of the PL. To enable reuse of requirements through configuration at the requirements engineering phase, it is important to provide a methodology (with tool support) to help practitioners to systematically and automatically develop structured and configuration-ready PL requirements repositories. Such a repository can cost-effectively facilitate the development of requirement repositories specific to individual products, i.e., individual systems. In addition, configurations to the repository at the requirements engineering phase of developing a system are part of its complete configurations and can be naturally carried on to downstream product configuration phases such as the design level configuration phase. A complete set system configurations can then be systematically obtained and managed. In this paper, we propose a methodology with tool support, named as Zen-ReqConfig, which is built on existing model-based technologies, natural language processing, and similarity measure techniques, for developing PL requirement repositories and facilitating requirements configuration. Zen-ReqConfig first automatically devises a hierarchical structure for a PL requirements repository. Then, it automatically identifies variabilities in textual requirements. Based on the developed configuration-ready PL requirements repository, it can then facilitate the configuration of products/systems at the requirements level. Zen-ReqConfig relies on two types of variability modeling techniques: cardinality-based feature modeling (CBFM) and a UML-based variability modeling methodology (named as SimPL). Both CBFM and SimPL have been used to address real-world variability modelling problems. To gain insights on the performance of Zen-ReqConfig, we evaluated it with five case studies and experimented with two different similarity measures and two different modelling methods: SimPL

and CBFM. Results show that Zen-ReqConfig performed better when it is combined with the Jaro similarity measure. When Zen-ReqConfig is integrated with Jaro, it can (1) structure PL textual requirements under the most fit match criterion with high precision and recall, over 95% for both CBFM and SimPL; (2) identify variabilities in textual requirements under the most fit match criterion, with the average precision over 97% for SimPL and CBFM, and with the average recall over 94% for both SimPL and CBFM; and (3) generate repository structures within 1 second; 4) and allocate a requirement to the repository within 2 seconds on average. When looking into the impact of the two modelling methods on the performance of Zen-ReqConfig, we did not observe practical differences between SimPL and CBFM, implying that Zen-ReqConfig works well with both SimPL and CBFM.

Pointer to the original paper:  
<https://link.springer.com/article/10.1007/s10270-017-0641-6>

## CCS CONCEPTS

- Software and its engineering → Software product lines.

## KEYWORDS

software product lines, requirements engineering, model-based engineering

### ACM Reference Format:

Yan Li, Tao Yue, Shaukat Ali, and Li Zhang. 2019. Enabling Automated Requirements Reuse and Configuration. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342370>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7138-4/19/09.

<https://doi.org/10.1145/3336294.3342370>

# Formal Foundations for Analyzing and Refactoring Delta-Oriented Model-Based Software Product Lines

Christopher Pietsch

Udo Kelter

cpietsch@informatik.uni-siegen.de

kelter@informatik.uni-siegen.de

University of Siegen

Siegen, Germany

Timo Kehrer

timo.kehrer@informatik.hu-berlin.de

Humboldt-Universität zu Berlin

Berlin, Germany

Christoph Seidl

c.seidl@tu-braunschweig.de

Technische Universität Braunschweig

Braunschweig, Germany

## ABSTRACT

*Model-Based Software Product Line (MBSPL) Engineering* combines Model-Based Software Engineering (MBSE) and Software Product Line (SPL) Engineering by specifying variability in models and generating model variants as products of an MBSPL. Delta Modeling (DM) is a transformational approach for implementing MBSPLs by adding, removing or modifying model elements through delta modules to activate individual features. To date, the applicability of DM to real-world MBSPLs is severely hindered due to the resulting complex network of interrelated delta modules in which errors are hard to identify and fix without unintentionally harming overall consistency. To address this challenge, we present a set of analyses to identify problems in a network of delta modules as well as a construction kit to assemble refactorings to remedy these problems and simplify the network. We give a modeling-language independent formalization of delta modules based on graph transformation concepts. This is the basis for our analyses which, in turn, build the basis for our refactorings to prevent unintended side-effects.

## CCS CONCEPTS

- Software and its engineering → Model-driven software engineering; Automated static analysis; Software product lines.

## KEYWORDS

Model-based Software Product Line Engineering, Delta Modeling, Graph Transformation, Analysis, Refactoring

### ACM Reference Format:

Christopher Pietsch, Udo Kelter, Timo Kehrer, and Christoph Seidl. 2019. Formal Foundations for Analyzing and Refactoring Delta-Oriented Model-Based Software Product Lines. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3336294.3336299>

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336299>

## 1 INTRODUCTION

*Software Product Line Engineering (SPL)* [23] provides methods for developing variant-rich systems. A Software Product Line (SPL) is realized in terms of functionality shared by all products (*commonalities*) and individual to a subset of products (*variabilities*). Conceptually, configurable functionality is commonly represented in terms of features. The configuration logic to determine valid combinations of features (i.e., configurations) is specified in a *Feature Model (FM)* [3]. On realization level, features are implemented in reusable artifacts. If fully automated, a product of an SPL may be generated upon request for any configuration.

With the advent of *Model-Based Software Engineering (MBSE)*, models have become primary development artifacts. *Model-Based Software Product Line (MBSPL) Engineering* combines MBSE and SPL by subjecting implementation models to variability and generating model variants as products of an MBSPL. *Delta Modeling (DM)* is a transformational approach for realizing MBSPLs [26, 28]. In essence, an MBSPL is implemented by a *core model*, representing one concrete product, and a set of interrelated *delta modules* defining transformations to realize individual features and their interactions by adding, removing and modifying model elements.

The main task during MBSPL implementation is thus to develop a core model and a set of delta modules that are able to generate all products of the MBSPL. To ensure consistent application of the transformations defined by delta modules, a deterministic application order for delta modules must be defined. This is a challenging task since delta modules may depend on each other or be in conflict if applied together. Existing tools [20, 32, 33, 39] offer no or only limited support for detecting and managing such interrelations. Dependencies must be maintained manually by specifying a partial order on sets of delta modules. For detecting inappropriate dependencies and conflicts, theoretically, all products would have to be generated, which is infeasible due to the combinatorial explosion of configurations. When a conflict occurs, the involved delta modules must be revised accordingly, which is challenging due to unexpected side effects. Thus, appropriate means are required to cope with and simplify the complexity of the network of delta modules. To fill this gap, we present the following contributions:

- a. We formalize transformations of delta modules as graph transformations with well-defined operational semantics,
- b. we exploit this formalization to statically analyze the transformations defined by delta modules for problematic interrelations
- c. we present a construction kit to assemble refactorings to remedy detected problems and to simplify the network of delta modules,

- d. we demonstrate our analysis functions and refactorings using a case study from the automation domain.

We implemented and integrated the proposed techniques in a development environment for MBSPLs known as SiPL [20], which is realized on top of the Eclipse Modeling technology stack. SiPL provides extension points to integrate analyses and refactorings on sets of interrelated delta modules, and provides further tooling facilities to visualize the analysis results and to trigger refactorings. A general quality assurance process which exploits such tool functionality is presented in [21]. We kindly refer to [20–22] for such tooling and process-related aspects, while, in this paper, we focus on the formal foundations of our techniques.

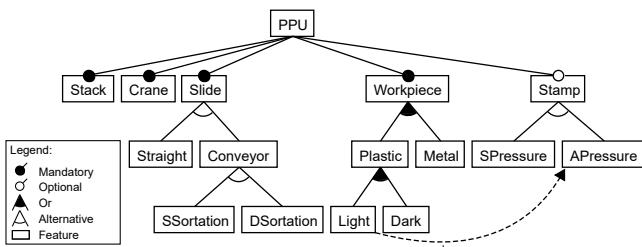
The paper is structured as follows. Section 2 recalls the main concepts of DM and introduces a running example. Section 3 formalizes delta modules based on graph transformation concepts. Section 4 and Section 5 introduce our analysis functions and refactoring operations, respectively. Section 6 demonstrates and validates their usage to implement and refactor an MBSPL. Section 7 discusses related work and Section 8 concludes the paper.

## 2 BACKGROUND AND MOTIVATION

We motivate the main challenges of applying DM within an MBSPL by a running example, which is an excerpt from our case study from the automation domain, the *Pick and Place Unit (PPU)* [37].

Figure 1 shows the *problem space* of the PPU MBSPL in terms of a FM [3, 13], which is used for configuring the control software of the PPU in accordance with its physical setup. The PPU always consists of a STACK serving as an input storage of WORKPIECES, a SLIDE serving as an output storage, and a CRANE transporting a workpiece from the stack to the slide. There are different types of workpieces, namely METAL and PLASTIC workpieces, the latter are further distinguished into LIGHT and DARK. Optionally, a workpiece can be labeled by a STAMP before transporting it to the slide. The features WORKPIECE, SLIDE and STAMP define *variation points* where one or more options can be chosen. Workpieces can be merely pushed out onto a ramp (STRAIGHT) or they can be sorted according to different criteria (SSORTATION, DSORTATION) using a CONVEYOR with multiple ramps. The feature STAMP offers a standard (SPRESSURE) and an adaptive (APPRESSURE) variant, of which one can be chosen. Light plastic workpieces are always stamped and require a stamp with an adaptive pressure.

In DM, the *solution space* is realized in terms of a core model representing a valid product of the MBSPL and a set of delta modules. A delta module consists of calls to predefined *delta operations* transforming the core model to yield another product. For the sake of clarity, we refer to calls to delta operations as *delta actions*.



**Figure 1:** Problem space of the PPU MBSPL.

To map the problem space onto the solution space, each delta module is equipped with an *application condition*, which is a propositional expression over features of the FM. Hence, an application condition specifies which features (or combinations thereof) have to be selected in a configuration such that a delta module's delta actions are to be applied.

Figure 2 shows an example of the solution space of the PPU MBSPL, which uses state machines as implementation models. The depicted excerpt specifies the system's emergency behavior for the configuration  $P1 = \{\text{PPU, Stack, Crane, Slide, Straight, Workpiece, Metal}\}$ . The non-colored elements are part of the core model. Colored elements are added, removed or modified by the delta modules shown on the right hand side:

$\Delta_{Conveyor}$  specifies changes to be applied onto the core model when the corresponding feature is selected. It adds the state ConveyorStop as well as the transition t4 and modifies the guard of transition t2.

$\Delta_{Stamp}$  realizes the system's emergency behavior when the feature STAMP is selected. It adds the state EmergencyStopStamp, transitions t5/t6 and a guard to transition t5. Furthermore, it removes the state CraneStop and transition t3.

$\Delta_{ConveyorStamp}$  adds transition t7, removes transition t5 with its guard and modifies the guard of t2. It realizes the interaction of the corresponding features.

When generating a specific product, the involved delta modules are applied sequentially onto the core model. An application order must be chosen that is suitable for the dependencies between the delta modules. For instance,  $\Delta_{ConveyorStamp}$  must be applied after  $\Delta_{Conveyor}$  and  $\Delta_{Stamp}$ . Such dependencies are indicated by solid blue-colored unidirectional arrows. The delta action of  $\Delta_{ConveyorStamp}$  adding the transition  $t_7$  can only be applied after the respective source and target state are created, which is achieved by delta actions of  $\Delta_{Conveyor}$  and  $\Delta_{Stamp}$ .

To date, such dependencies must be identified manually, which is time-consuming and prone to errors. Moreover, evolutionary changes may invalidate the identified dependencies or may be inconsistent with the dependencies specified in the FM, e.g., when a feature and its associated delta modules are deleted.

Furthermore, delta modules may be in conflict, i.e., their application in an arbitrary order does not commute or even fail. For instance, the delta module  $\Delta_{Conveyor}$  and  $\Delta_{ConveyorStamp}$  are in conflict, which is illustrated by the dotted red-colored bidirectional arrow. Both modify the guard of transition  $t2$  and the result depends on the application order. A developer must be aware of such a conflict when specifying an application order. In this case, the conflict can be resolved automatically by the order implied through the dependency between the involved delta modules.

However, there are also conflicts for which no sequential order exists in which the involved delta modules can be applied without errors. For instance, the delta modules  $\Delta_{Conveyor}$  and  $\Delta_{Stamp}$  can be applied correctly in isolation, i.e., if only one of the features CONVEYOR and STAMP is selected. No appropriate order can be found when both features are selected, which is indicated by the solid red-colored bidirectional arrow.  $\Delta_{Conveyor}$  contains a delta action adding transition t4 while  $\Delta_{Stamp}$  contains a delta action deleting the transition's source state CraneStop. One main reason

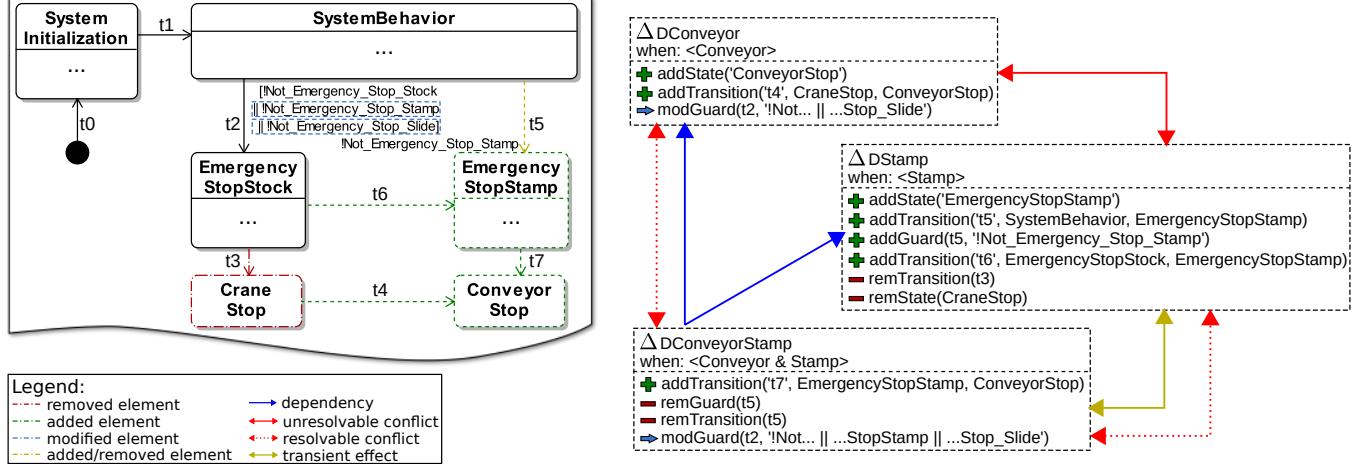


Figure 2: Excerpt of the solution space of the PPU MBSPL.

for such unresolvable conflicts is the *optional feature problem* [14], which occurs if the implementation of an optional feature interferes with the implementation of another feature.

There are several options to remove such unresolvable conflicts. For instance, it is possible to realize two delta modules for the feature CONVEYOR with mutually exclusive application conditions CONVEYOR & !STAMP and CONVEYOR & STAMP. However, this solution leads to replication of delta actions, referred to as *duplicates* in the sequel, similar to the *redundant code smell* [9]. When the conveyor's behavior changes over time, this creates an additional maintenance burden as all replications must be identified and revised. Another solution would be a further delta module  $\Delta_{invConveyor}$  with the application condition CONVEYOR & STAMP that partially reverts the effect of  $\Delta_{Conveyor}$ . We use the term *transient effect* when the effect of a delta action is reverted by a subsequent delta action. Such an effect is illustrated by the yellow-colored transition  $t_5$  in Figure 2. The delta module  $\Delta_{Stamp}$  adds transition  $t_5$  along with its guard while both elements are removed by the delta module  $\Delta_{ConveyorStamp}$ . Regarding our example for solving the optional feature problem by introducing  $\Delta_{invConveyor}$ ,  $\Delta_{Stamp}$  must be applied after  $\Delta_{Conveyor}$  and  $\Delta_{invConveyor}$ , otherwise its application would still fail. The identification of such *conditional dependencies* is challenging. Furthermore, transient effects may lead to *dead delta actions*, i.e., the effect of a delta action is always reverted by another delta action. This is similar to the *dead code smell* [9]. To avoid replications or transient effects when solving the optional feature problem, we can manually extract the conflicting delta actions into two separate delta modules having mutually exclusive application conditions, i.e., CONVEYOR & !STAMP and !CONVEYOR & STAMP. Furthermore, we have to specify the corresponding dependencies to the origin delta modules.

Up to now, there is no guarantee that the overall solution space is correct, i.e., no errors occur during assembling products due to unfulfilled dependencies or conflicting delta actions. Even when validating all products separately, which is impossible for complex MBSPLs, we would have to manually identify and fix the problems in the respective delta modules, which may introduce further errors. One reason for the missing automation of these tasks is the lack of a formal and uniform understanding of the effects of delta operations

for a given modeling language. To date, language-specific delta operations are engineered following an imperative approach. Their effect can hardly be analyzed in a uniform way [32, 33, 39] as it largely depends on the used programming language. Theory-based formalizations for delta modeling exist, but their analysis capabilities focus on properties such as correct typing of generated source artifacts and they do not offer refactoring support to mitigate design problems and errors [5, 7, 27].

To remedy this shortcoming, we provide a formalization of delta operations describing their effects in a uniform way. This builds the foundation for our generic analysis functions and integrates seamlessly with the practice of MBSE.

### 3 FORMALIZATION OF DELTA MODELING

In this section, we formalize DM based on graph transformation concepts [8]. We briefly recall the needed concepts in Section 3.1 before we formalize delta operations and delta actions in Section 3.2.

#### 3.1 Graph Transformation

A model is represented as *abstract syntax graph* (ASG) that is typed over the meta-model of the modeling language. Formally, a meta-model is represented as an attributed type graph  $TG$  (for details, see [4, 11]). An instance graph over  $TG$ , representing the ASG of a model, is a graph  $G$  equipped with a structure-preserving mapping  $G \rightarrow TG$  assigning every element in  $G$  its type in  $TG$ . The way we handle node attributes in an instance graph is to consider them as edges of a special kind referring to data values, while attributes declared by node types of a type graph are represented as special edges referring to data types. We write  $e := v$  to express that an attribute  $e$  has a value  $v$  which is compatible to the data type defined by the attribute declaration.

A graph transformation rule  $r(x_1, \dots, x_n) : L \rightarrow R$ , formally graph inclusions  $L \leftarrow K \rightarrow R$  as shown in the upper part of Figure 3, declaratively defines a model transformation by pre and post condition graphs  $L$  and  $R$ , the *left-hand* and *right-hand* side of the rule.  $L \setminus R$ ,  $L \cap R (= K)$  and  $R \setminus L$  specify the model elements to be *deleted*, *preserved* and *created* by the rule, respectively. The modification of an attribute value is considered as deletion and (re-) creation of the

attribute. Together, we call the deletions and creations the *change actions* specified by the rule.

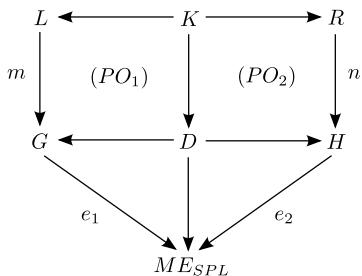
Given an instance graph  $G$ , a rule can be applied if there is a match  $m : L \rightarrow G$  such that  $L$  is isomorphic to a subgraph of  $G$ . The effect of a rule application at match  $m$  is defined by removing  $m(L \setminus R)$  from  $G$  and adding a copy of  $R \setminus L$ , yielding the modified graph  $H$  and the so-called *co-match*  $n : R \rightarrow H$ . The formal construction of such a transformation is a double-pushout (DPO), which is shown in the diagram in Figure 3 with pushouts  $(PO_1)$  and  $(PO_2)$  in the category of typed graphs. Graph  $D$  is the intermediate graph after removing  $m(L \setminus R)$ , and  $H$  is constructed as gluing of  $D$  and  $R$  along  $K$  (see [8]). The context of a rule application can be determined by passing context elements via parameters  $x_1, \dots, x_n \in L \cup R$ . Parameters refer to rule elements, and passing concrete arguments yields a so-called *pre-match*, a partial mapping  $m_p : L \rightarrow G$ .

Figure 4 shows graph transformation rules typed over the UML meta-model. We use the notation of the model transformation language Henshin [2] describing the left- and right-hand side of a rule in a single unified graph. The left-hand side comprises all elements stereotyped by «delete» and «preserve». The right-hand side contains all elements stereotyped by «preserve» and «create». Rule `addTransition(r:Region, src:Vertex, tgt:Vertex)` creates a Transition  $t$  between source and target states given by the parameters  $src$  and  $tgt$ , respectively. Transition  $t$  is inserted into the Region passed as parameter  $r$ . Values of properties of the newly created transition  $t$ , e.g., its name, are passed as additional parameters. Rule `removeState(s:State)` deletes a State  $s$  that is passed to the rule as parameter. No parameter for the region is needed as it can be uniquely determined as the container of the passed state.

### 3.2 Delta Operations and Delta Actions

We assume that all model elements referred to by delta actions can be uniquely identified. To that end, we use a generic graph structure superimposing all model elements used in the MBSPL. Analogously to instances graphs, such a superimposed ASG, in the sequel referred to as  $ME_{SPL}$ , is typed over the meta-model of the respective modeling language but relaxes multiplicity constraints for edges and attribute values.

A delta operation is a conventional graph transformation rule, while a delta action is defined as a specific kind of graph transformation over  $ME_{SPL}$ , according to the diagram shown in Figure 3. The original graph  $G$  and the modified graph  $H$  of the



**Figure 3: Formalization of delta operations and delta actions based on graph transformation concepts following the double-pushout approach.**

transformation are isomorphic to subgraphs of the superimposed ASG  $ME_{SPL}$ , indicated by the graph inclusions  $e_1$  and  $e_2$ , respectively. The left-hand side rule elements and right-hand side rule elements are embedded into  $ME_{SPL}$  through compositions  $e_1 \circ m$  and  $e_2 \circ n$ , respectively. For convenience reasons, we define a mapping relation  $o = (e_1 \circ m) \cup (e_2 \circ n)$  to indicate occurrences of rule elements in the superimposed ASG. We use the notation  $\delta_{op}(x_1, \dots, x_n) : L_{op} \rightarrow R_{op}$  (or  $\delta_{op}$  for short) to refer to delta operations. A delta action  $\delta_{op}^o$  is treated as a pair  $\delta_{op}^o = (\delta_{op}, o_{op})$  with  $o_{op} : L_{op} \cup R_{op} \rightarrow ME_{SPL}$  indicating its embedding into the superimposed ASG. A delta module  $\Delta$  is a tuple  $\Delta = (DA, \varphi)$  where  $DA$  is a partially ordered set of delta actions and  $\varphi$  is a propositional formula over a subset of the features of the FM specifying the application condition of  $\Delta$ .

Figure 4 shows an example where the elements of the superimposed ASG  $ME_{SPL}$  in the center are colored according to delta actions, the delta operations of which are depicted in the upper and lower part of the figure. Occurrences of rule elements are illustrated as dashed and dotted unidirectional arrows (mappings of edges and attributes are omitted for the sake of readability). For instance, the node  $s \in L_{remState} \cup R_{remState}$  of the delta operation  $\delta_{remState}$  is mapped onto the state `CraneStop`, i.e.,  $o_{remState}(s) = \text{CraneStop:State}$ , which is part of the core model. The state serves as parameter of the delta operation and is part of the pre-match. The mapping of the node  $r \in L_{remState} \cup R_{remState}$  of the same delta operation, i.e.,  $o(r) = \text{:Region}$ , is implicitly derived when completing the partial match. Analogously, the node  $r \in L_{addState} \cup R_{addState}$  of the delta operation `addState` is mapped via parameter assignment, i.e.,  $o_{addState}(r) = \text{:Region}$ . The rule node  $s \in R_{addState}$  of this operation does not yet exist in the current state of the superimposed ASG. It is added along with its incident edges and attributes, yielding the state `ConveyorStop:State` and edges of type `subvertex` and `container`.

Technically, the superimposed ASG, delta modules, delta operations and their embeddings into the superimposed ASG in terms of delta actions are constantly managed in the background by the SiPL development environment, which implements our approach. Furthermore, sets of delta operations for a given modeling language are generated from the language's meta-model using the approach and supporting tool presented in [15, 24]. The generated operations are sound in the sense that they preserve consistency constraints defined by the meta-model, and parameter lists of an operation are complete in the sense that any assignment yields a pre-match that is extensible to a complete match deterministically. Moreover, a generated operation set is complete in the sense that any model modification can be expressed using operations available in the set.

## 4 STATIC ANALYSIS FOR DETECTING PROBLEMATIC INTERRELATIONS

Based on our formal definitions, we define a set of analyses to determine various interrelations between delta actions statically, i.e., without generating any products. Table 1 summarizes our definitions for those interrelations with a brief explanation. We further aggregate detected interrelations to relations between delta modules, which are validated against the FM to determine if they actually occur in the overall MBSPL.

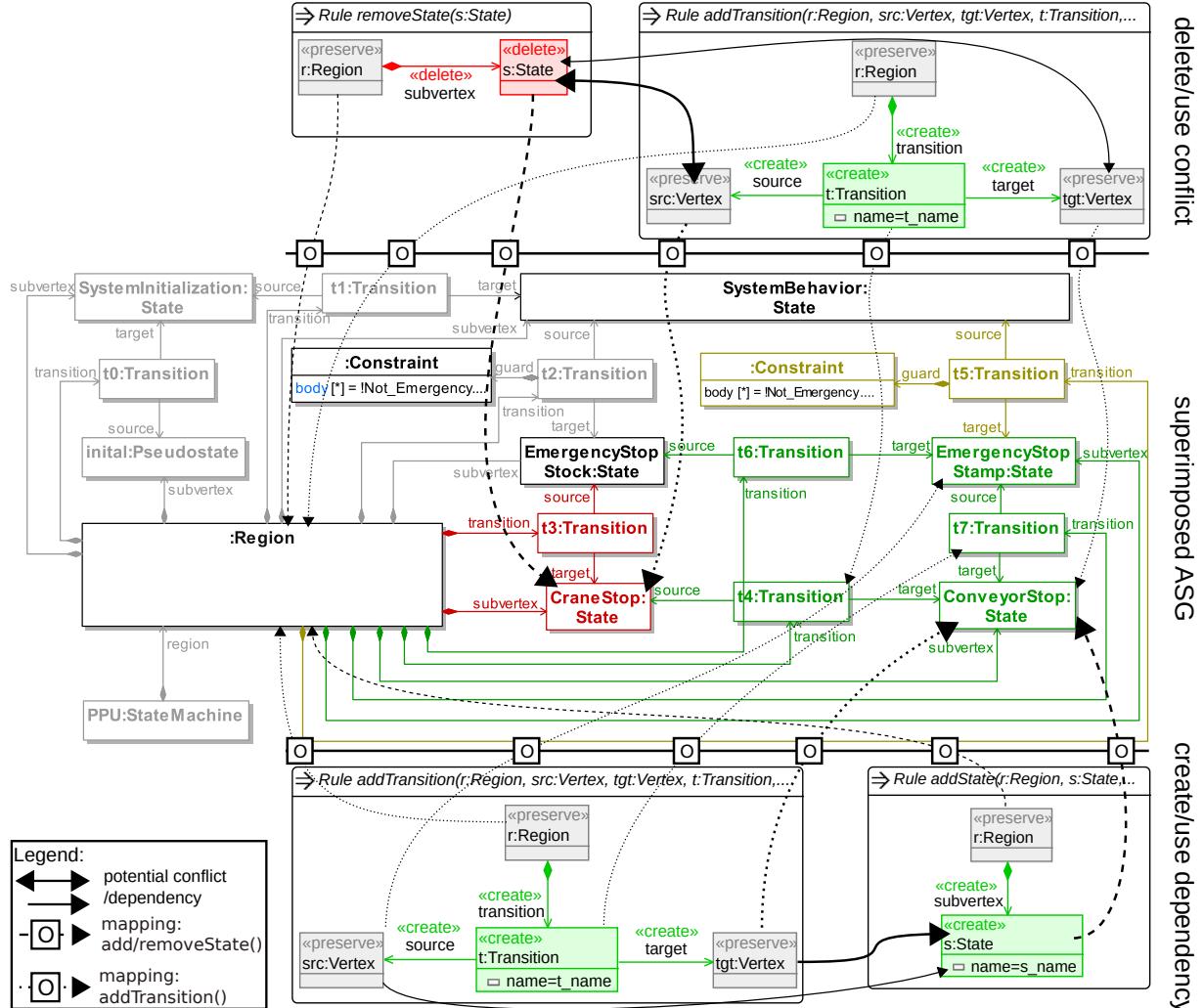


Figure 4: Conflict and dependency detection illustrated using exemplary excerpts of the PPU MBSPL implementation.

## 4.1 Dependency Detection

As a prerequisite, all pairs of delta operations of the overall set of delta operations for a given modeling language are analyzed for *potential dependencies* using critical pair analysis [8, 11, 16]. A delta operation  $\delta_{op_2}$  potentially depends on another delta operation  $\delta_{op_1}$  if at least one rule element  $e_2 \in \delta_{op_2}$  potentially depends on a rule element  $e_1 \in \delta_{op_1}$  regarding their change actions. We distinguish two kinds of potential dependencies:

**Create/Use:**  $\delta_{op_1}$  creates a node/edge  $e_1$  that is potentially matched by a node/edge  $e_2$  of  $\delta_{op_2}$ , i.e.,  $e_1 \in (R_{op_1} \setminus L_{op_1})$  and  $e_2 \in L_{op_2}$ .

**Change/Use:**  $\delta_{op_1}$  changes a value of an attribute  $e_1$  to  $v$  that is potentially matched by an attribute  $e_2$  of  $\delta_{op_2}$ , i.e.,  $(e_1 := v) \in (R_{op_1} \setminus L_{op_1})$  and  $(e_2 := v) \in L_{op_2}$ .

The critical pair analysis is done only once for the overall set of delta operations. To check if a potential dependency is an actual one, we define the function  $depends(\delta_{op_1}^o, \delta_{op_2}^o)$  as follows. Given two delta actions  $\delta_{op_1}^o = (\delta_{op_1}, o_1)$  and  $\delta_{op_2}^o = (\delta_{op_2}, o_2)$  using delta operations  $\delta_{op_1}$  and  $\delta_{op_2}$ ,  $\delta_{op_2}^o$  actually depends on  $\delta_{op_1}^o$  if the following conditions hold:

1. There are at least two rule elements  $e_1 \in \delta_{op_1}$  and  $e_2 \in \delta_{op_2}$  such that there is a potential dependency between  $e_1$  and  $e_2$ .
2. The rule elements  $e_1$  and  $e_2$  are mapped onto the same element in  $ME_{SPL}$ , i.e.,  $o(e_1) = o(e_2)$ .

Figure 4 shows an example of an actual Create/Use dependency. The lower part shows the delta operations  $\delta_{addState}$  and  $\delta_{addTrans}$  used by the delta actions  $\delta_{addState}^o = (\delta_{addState}, o_{addState})$  and  $\delta_{addTrans}^o = (\delta_{addTrans}, o_{addTrans})$ . (1)  $\delta_{addTrans}$  potentially depends on  $\delta_{addState}$ , which is illustrated by solid unidirectional arrows.  $\delta_{addState}$  creates a state  $s$  that is potentially used as  $src$  or  $tgt$  of  $\delta_{addTrans}$ . (2) The rule elements  $tgt \in \delta_{addTrans}$  and  $s \in \delta_{addState}$  are mapped onto the same state ConveyorStop, i.e.,  $o_{addTrans}(tgt) = o_{addState}(s)$ . The delta action  $\delta_{addTrans}^o$  can be applied only after the delta action  $\delta_{addState}^o$  and thus depends on the latter.

A delta module  $\Delta_2 = (DA_2, \varphi_2)$  depends on a delta module  $\Delta_1 = (DA_1, \varphi_1)$  if at least one delta action  $\delta_{op_2}^o \in DA_2$  depends on a delta action  $\delta_{op_1}^o \in DA_1$ . The function  $depends(\Delta_1, \Delta_2)$  returns the set of pairs of delta actions  $(\delta_{op_1}^o, \delta_{op_2}^o) \in \Delta_1 \times \Delta_2$  where  $\delta_{op_2}^o$

Kind	Definition	Explanation
Dependency	$\delta_{op_2}^o$ depends on $\delta_{op_1}^o$ if $\delta_{op_2}^o$ can only be applied after $\delta_{op_1}^o$ .	$\delta_{op_1}^o$ creates a model element that is needed by $\delta_{op_2}^o$ , or $\delta_{op_1}^o$ changes an attribute value such that an initially unfulfilled precondition of $\delta_{op_2}^o$ is fulfilled.
Conflict	The delta actions $\delta_{op_1}^o$ and $\delta_{op_2}^o$ are in conflict if they cannot be applied together or their application in both orders would lead to different results.	$\delta_{op_1}^o$ deletes a model element which is needed by $\delta_{op_2}^o$ or changes an attribute value such that a precondition of $\delta_{op_2}^o$ is not fulfilled any more, or both modify the same attribute by setting different values.
Duplicate	The delta actions $\delta_{op_1}^o$ and $\delta_{op_2}^o$ yield the same effect under the same condition.	$\delta_{op_1}^o$ and $\delta_{op_2}^o$ create or delete the same element, or modify the same attribute by setting the same value.
Transient Effect	The application of the delta action $\delta_{op_2}^o$ removes the effect of the delta action $\delta_{op_1}^o$ .	$\delta_{op_1}^o$ creates a model element that is deleted by $\delta_{op_2}^o$ , or $\delta_{op_1}^o$ deletes a model element or changes the value of an attribute which is later restored by $\delta_{op_2}^o$ .

Table 1: Interrelations between delta actions.

depends on  $\delta_{op_1}^o$ . We use the notation  $\delta_{op_1}^o < \delta_{op_2}^o$  as a short form of  $(\delta_{op_1}^o, \delta_{op_2}^o) \in \text{depends}(\Delta_1, \Delta_2)$ .

To check if the dependency is not fulfilled in the overall solution space, we check if the following condition holds:  $FM \wedge \neg(\varphi_2 \implies \varphi_1)$ . An unfulfilled dependency must be resolved, e.g., by changing the application condition of an involved delta module.

## 4.2 Conflict Detection

Analogously to potential dependencies, we analyze all pairs of delta operations for *potential conflicts* using critical pair analysis. A pair of delta operations  $(\delta_{op_1}, \delta_{op_2})$  potentially conflicts if at least one pair of rule elements  $e_1 \in \delta_{op_1}$  and  $e_2 \in \delta_{op_2}$  potentially conflicts regarding their change actions. Again, we distinguish two kinds of potential conflicts:

**Delete/Use:**  $\delta_{op_1}$  deletes a node/edge  $e_1$  that potentially is matched by a node/edge  $e_2$  of  $\delta_{op_2}$ , i.e.,  $e_1 \in (L_{op_1} \setminus R_{op_1})$  and  $e_2 \in L_{op_2}$ .

**Change/Use:**  $\delta_{op_1}$  changes the value  $v$  of an attribute  $e_1$  that potentially is matched by an attribute  $e_2$  of  $\delta_{op_2}$ , i.e.,  $(e_1 := v) \in (L_{op_1} \setminus R_{op_1})$  and  $(e_2 := v) \in L_{op_2}$ .

We define the function  $\text{conflicts}(\delta_{op_1}^o, \delta_{op_2}^o)$  that checks if a potential conflict is an actual one as follows. Given two delta actions  $\delta_{op_1}^o = (\delta_{op_1}, o_1)$  and  $\delta_{op_2}^o = (\delta_{op_2}, o_2)$ , they are actually conflicting if the following conditions hold:

1. There are at least two rule elements  $e_1 \in \delta_{op_1}$  and  $e_2 \in \delta_{op_2}$  such that there is a potential conflict between  $e_1$  and  $e_2$ .
2. The rule elements  $e_1$  and  $e_2$  are mapped onto the same element in  $ME_{SPL}$ , i.e.,  $o(e_1) = o(e_2)$ .

Figure 4 shows an example of an actual *Delete/Use* conflict. The upper part shows the delta operations  $\delta_{remState}$  and  $\delta_{addTrans}$  used by the delta actions  $\delta_{remState}^o = (\delta_{remState}, o_{remState})$  and  $\delta_{addTrans}^o = (\delta_{addTrans}, o_{addTrans})$ . (1) They are potentially in conflict, illustrated by solid bidirectional arrows. The application of  $\delta_{remState}$  deletes a state  $s$  that is potentially matched by  $src$  or  $tgt$  of  $\delta_{addTrans}$ . (2) Rule elements  $s \in \delta_{remState}$  and  $src \in \delta_{addTrans}$ , which are in a potential conflict, are mapped onto the same state  $CraneStop$ , i.e.,  $o_{remState}(s) = o_{addTrans}(src)$ . Both delta actions cannot be applied together and thus are in conflict.

However, two special kinds of conflicts can be resolved by choosing a suitable order in which the conflicting delta actions are executed. For instance, for a delta action  $\delta_{op_1}^o$  that deletes a transition between a source and target state as well as a delta action  $\delta_{op_2}^o$  that deletes the source state,  $\delta_{op_1}^o$  has to be executed before  $\delta_{op_2}^o$ . Such a conflict implicitly yields a dependency from  $\delta_{op_2}^o$  to  $\delta_{op_1}^o$  that is automatically determined. As another example, the application of two delta actions changing the same attribute value does not fail,

but leads to different results if applied in different orders. Hence, to avoid ambiguous results, a dependency must be specified manually.

Two delta modules  $\Delta_1 = (DA_1, \varphi_1)$  and  $\Delta_2 = (DA_2, \varphi_2)$  are in conflict if at least one delta action  $\delta_{op_1}^o \in DA_1$  conflicts with a delta action  $\delta_{op_2}^o \in DA_2$  that cannot be treated as dependency. The analysis function  $\text{conflicts}(\Delta_1, \Delta_2)$  returns a set of pairs of delta actions  $(\delta_{op_1}^o, \delta_{op_2}^o) \in \Delta_1 \times \Delta_2$  where  $\delta_{op_1}^o$  and  $\delta_{op_2}^o$  are in conflict. We use the notation  $\delta_{op_1}^o \not\leq \delta_{op_2}^o$  as a short form of  $(\delta_{op_1}^o, \delta_{op_2}^o) \in \text{conflicts}(\Delta_1, \Delta_2)$ .

To check if a conflict actually occurs in the overall solution space, we check if the following condition holds:  $FM \wedge (\varphi_1 \wedge \varphi_2)$ . A detected conflict must be resolved, e.g., by extracting the conflicting delta actions into separate delta modules having mutually exclusive application conditions.

## 4.3 Duplicate Detection

Two delta actions  $\delta_{op_1}^o = (\delta_{op_1}, o_1)$  and  $\delta_{op_2}^o = (\delta_{op_2}, o_2)$  are identical if the following conditions hold:

1. The delta actions use the same delta operation, i.e.,  $\delta_{op_1} = \delta_{op_2}$ .
2. Identical rule elements are mapped to the same elements in  $ME_{SPL}$ , i.e.,  $o(L_{op_1}) = o(L_{op_2})$  and  $o(R_{op_1}) = o(R_{op_2})$ .

The analysis function  $\text{duplicates}(\Delta_1, \Delta_2)$  returns all pairs  $(\delta_{op_1}^o, \delta_{op_2}^o) \in \Delta_1 \times \Delta_2$  of identical delta actions. We use the notation  $\delta_{op_1}^o = \delta_{op_2}^o$  as a short form of  $(\delta_{op_1}^o, \delta_{op_2}^o) \in \text{duplicates}(\Delta_1, \Delta_2)$ .

To check if the detected duplicates also occur in the overall solution space, we check the following condition:  $FM \wedge (\varphi_1 \wedge \varphi_2)$ . If this condition holds, duplicates must be eliminated, e.g., by extracting the respective delta actions into a new delta module. This way, we also identify conflicting delta actions that only occur due to duplicates. For instance, a delta action of a delta module deleting a transition conflicts with a delta actions of another delta module that deletes the same transition. This conflict would be resolved automatically by extracting the common delta actions. Even if both delta modules would never be applied together, i.e., the condition does not hold, extracting the duplicate delta actions may reduce maintenance efforts.

## 4.4 Detection of Transient Effects

Two delta actions  $\delta_{op_1}^o = (\delta_{op_1}, o_1)$  and  $\delta_{op_2}^o = (\delta_{op_2}, o_2)$  lead to transient effects if the following conditions hold:

1. Rule elements that are preserved by  $\delta_{op_1}$  and  $\delta_{op_2}$  must be mapped to the same model elements, i.e.,  $o(L_{op_1} \cap R_{op_1}) = o(L_{op_2} \cap R_{op_2})$ .

2. Rule elements that are deleted by  $\delta_{op_1}$  and created by  $\delta_{op_2}$  must be mapped to the same model elements, i.e.,  $o(L_{op_1} \setminus R_{op_1}) = o(R_{op_2} \setminus L_{op_2})$ .
3. Rule elements that are created by  $\delta_{op_1}$  and deleted by  $\delta_{op_2}$  must be mapped to the same model elements, i.e.,  $o(R_{op_1} \setminus L_{op_1}) = o(L_{op_2} \setminus R_{op_2})$ .

Two delta modules  $\Delta_1, \Delta_2$  produce transient effects if at least one delta action  $\delta_{op_2}^o \in \Delta_2$  removes the effect of a delta action  $\delta_{op_1}^o \in \Delta_1$ . Furthermore, we define the analysis function  $transients(\Delta_1, \Delta_2)$ , which returns all pairs of edit steps  $(\delta_{op_1}^o, \delta_{op_2}^o) \in \Delta_1 \times \Delta_2$  where  $\delta_{op_2}^o$  removes the effect of  $\delta_{op_1}^o$ . Such a transient effect should be resolved if it leads to a dead delta action, i.e., if the following condition is fulfilled:  $FM \wedge \neg(\varphi_1 \wedge \neg\varphi_2 \vee \neg\varphi_1 \wedge \varphi_2)$ .

Furthermore, we also identify conditional dependencies, i.e., only one dependency of a given delta module must be fulfilled, e.g., a delta module depends on another delta module due to a transient effect but can also be applied in isolation. An example has already been given in Section 2, when solving the optional feature problem by adding the delta module  $\Delta_{invConveyor}$ . Analogously, we also identify conflicts that occur only due to transient effects. For instance, the delta module  $\Delta_{Stamp}$  and  $\Delta_{ConveyorStamp}$  in Figure 2 are in conflict.  $\Delta_{Stamp}$  adds a guard to transition t5, which is removed by the delta module  $\Delta_{ConveyorStamp}$ . The transition, in turn, can only be removed when the guard is also removed, which is also done by the latter delta module yielding to a transient effect. Furthermore, the latter delta module depends on the former. This dependency guarantees that the transition and its guard are added first. This way, the involved delta modules are not in an unresolvable conflict due to the transient effects and the corresponding dependencies.

## 5 RESOLVING PROBLEMATIC INTERRELATIONS

*Refactorings* are an established technique to improve the maintainability by restructuring a software system without changing its observable behavior [9]. In SPL, refactorings may be related to the problem space, the solution space or both, while preserving or enhancing the behavior of the overall SPL [1, 30]. Table 2 shows a subset of the set of refactorings introduced by Schulze et al [29] that are related to the solution space of delta-oriented SPLs. For each refactoring, a typical situation is described along with its effect.

Identifying potential refactoring opportunities in complex SPLs is challenging without appropriate tool support [29]. Furthermore, the application of refactorings may lead to unexpected side effects and thus should not be applicable in such a case, or the developer should be informed before applying the refactoring. For instance, the refactoring *Extract Conflicting Delta Actions* should only be applicable if two delta modules are in conflict, i.e., the conflict must be detected beforehand. Furthermore, this refactoring could lead to unexpected side effects, e.g., when other delta actions depend on the conflicting ones. The refactoring *Merge Delta Modules with Equivalent Application Condition* should only be applicable if both delta modules are not in conflict. Furthermore, if one of the delta modules depends on the other one, this order must be considered when merging the delta modules. *Remove Dead Delta Action* induces that the effect of a delta action is always reverted by another delta

action. When removing the dead delta action, we also have to remove all successors to preserve behavior. The refactoring *Resolve Duplicated Delta Actions* should only be applicable for delta modules sharing at least one common delta action. However, its application could lead to dependency cycles, e.g., if a common delta action depends on a non-common delta action and a non-common delta action depends on a common delta action.

To support the identification of refactoring opportunities and potential side effects, we make use of our analysis facilities. We present a set of operations with well-defined preconditions that serve as refactoring construction kit (see Section 5.1) to assemble (complex) refactorings (see Section 5.2).

### 5.1 Refactoring Construction Kit

The following operations create a new delta module on the basis of two or more interrelated delta modules, or they revise an existing delta module. For each operation, a precondition is defined that must be fulfilled such that the operation is applicable.

$concat(\Delta_1, \dots, \Delta_n)$ . The operation creates a new delta module that contains all delta actions of the arguments and extends their partial order according to the dependencies between pairs of the given delta modules. This operation is applicable if there is (1) no conflict and (2) no duplicated delta action between pairs of arguments, i.e., for all pairs  $(\Delta_i, \Delta_j)$  with  $1 \leq i < j \leq n$ , the following conditions must hold:

1.  $conflicts(\Delta_i, \Delta_j) = \emptyset$
2.  $duplicates(\Delta_i, \Delta_j) = \emptyset$

The delta application condition  $\varphi_{new}$  of the new delta module is a conjunction of the application conditions of the arguments, i.e.,  $\varphi_{new} = \varphi_1 \wedge \dots \wedge \varphi_n$ .

$intersect(\Delta_1, \Delta_2)$ . The operation creates a delta module that contains the common delta actions of both arguments. To be applicable, (1) the delta modules must share at least one common delta action and (2) no dependency cycle must be introduced, i.e., the following conditions must hold:

1.  $ce(\Delta_1, \Delta_2) \neq \emptyset$ , where  $ce(\Delta_1, \Delta_2) = \{\delta_{op_1}^o \in \Delta_1 \mid \exists \delta_{op_2}^o \in \Delta_2 : \delta_{op_1}^o = \delta_{op_2}^o\}$  denotes the “common” delta actions in  $\Delta_1$ <sup>1</sup>.
  2.  $(ncPre(\Delta_1, \Delta_2) = \emptyset \vee ncSuc(\Delta_1, \Delta_2) = \emptyset) \wedge (ncPre(\Delta_2, \Delta_1) = \emptyset \vee ncSuc(\Delta_2, \Delta_1) = \emptyset)$  where  $ncPre(\Delta_1, \Delta_2) = \{\delta_{op_1}^o \in ce(\Delta_1, \Delta_2) \mid \exists \delta_{op_2}^o \in nce(\Delta_1, \Delta_2) : \delta_{op_2}^o < \delta_{op_1}^o\}$  is the set of common delta actions that depend on a non-common preceding one.
- $ncSuc(\Delta_1, \Delta_2) = \{\delta_{op_1}^o \in ce(\Delta_1, \Delta_2) \mid \exists \delta_{op_2}^o \in nce(\Delta_1, \Delta_2) : \delta_{op_1}^o < \delta_{op_2}^o\}$  is the set of common delta actions having a dependent non-common delta action.
- $nce(\Delta_1, \Delta_2) = \Delta_1 \setminus ce(\Delta_1, \Delta_2)$  are the “non-common” delta actions of  $\Delta_1$ .

The delta application condition  $\varphi_{new}$  of the new delta module is a disjunction of the application conditions of the arguments  $\Delta_1$  and  $\Delta_2$ , i.e.,  $\varphi_{new} = \varphi_1 \vee \varphi_2$ .

<sup>1</sup>More precisely, the representatives of the common delta actions of  $\Delta_1$  and  $\Delta_2$  in  $\Delta_1$

Refactoring	Typical Situation	Effect
<i>Extract Conflicting Delta Actions</i>	Two delta modules that are in conflict shall be made consistent.	Extracts the conflicting delta actions into separate delta modules with mutually exclusive application conditions.
<i>Merge Delta Modules with Equivalent Application Conditions</i>	Two delta modules have an equivalent application condition.	Moves the delta actions of one delta module into the other and remove the empty delta module.
<i>Remove Dead Delta Action</i>	The effect of a delta action is always reverted by another delta action.	Removes the dead delta action and its successors i.e., all delta actions that depends on the dead delta action.
<i>Resolve Duplicated Delta Actions</i>	Two delta modules share at least one common delta action.	Extracts the common delta actions into a new one with a combined disjunctive application condition.

Table 2: Refactorings for DM

*intersectConflict*( $\Delta_1, \Delta_2$ ). The operation takes two conflicting delta modules as input and returns a new delta module that contains the delta actions of  $\Delta_1$  that are in conflict with a delta action of  $\Delta_2$  and all delta actions directly or transitively depending on a conflicting one. To be applicable (1) a conflict must exist and (2) there is at least one delta action in  $\Delta_1$  that does not depend on the conflicting delta action (otherwise we would simply replicate  $\Delta_1$ ), i.e. the following conditions must hold:

1.  $c(\Delta_1, \Delta_2) \neq \emptyset$ , where  $c(\Delta_1, \Delta_2) = \{\delta_{op_1}^o \in \Delta_1 \mid \exists \delta_{op_2}^o \in \Delta_2 : \delta_{op_1}^o \not\leq \delta_{op_2}^o\}$  is the set of delta actions in  $\Delta_1$  that are in conflict with delta actions in  $\Delta_2$ .
2.  $\Delta_1 \setminus (c(\Delta_1, \Delta_2) \cup cSuc(\Delta_1, \Delta_2)) \neq \emptyset$ , where  $cSuc(\Delta_1, \Delta_2) = \{\delta_{op_1}^o \in \Delta_1 \mid \exists \delta_{op_2}^o \in c(\Delta_1, \Delta_2) : \delta_{op_2}^o < \delta_{op_1}^o\}$  is the set of delta actions in  $\Delta_1$  which depend on a conflicting delta action.

The delta application condition  $\varphi_{new}$  of the new delta module is obtained from the application conditions of  $\Delta_1$  and  $\Delta_2$  as  $\varphi_{new} = \varphi_1 \wedge \neg \varphi_2$ .

*minus*( $\Delta_1, \Delta_2$ ). The operation creates a delta module that contains the set of delta actions contained in  $\Delta_1$  but not in  $\Delta_2$ . Analogously to *intersect*, for the operation to be applicable (1) the delta modules must share at least one delta action and (2) no dependency cycle must be introduced, i.e., the following conditions must hold:

1.  $ce(\Delta_1, \Delta_2) \neq \emptyset$
2.  $ncPre(\Delta_1, \Delta_2) = \emptyset \vee ncSuc(\Delta_1, \Delta_2) = \emptyset$

The delta application condition  $\varphi_{new}$  of the new delta module is equal to the application conditions of  $\Delta_1$ , i.e.,  $\varphi_{new} = \varphi_1$ .

*purge*( $\Delta$ ). The operation revises a delta module by removing all delta actions that lead to transient effects and is only applicable if the delta module contains transient effects, i.e., the following condition must hold:  $transients(\Delta_1, \Delta_1) \neq \emptyset$ .

## 5.2 Assembling Refactorings

The individual operations of the construction kit can, e.g., be assembled to the refactorings in Table 2. To check if a refactoring is applicable and does not lead to any unexpected side effect, we sequentially check the preconditions of the respective operations.

*Extract Conflicting Actions*. For extracting conflicting delta actions we assemble the operations *intersectConflict* and *minus*:

1. Check preconditions of the operations *intersectConflict*( $\Delta_1, \Delta_2$ ) and *intersectConflict*( $\Delta_2, \Delta_1$ ). If any precondition is unfulfilled, the refactoring is unavailable for the given delta modules.
2. Call the operations yielding the delta modules  $\Delta_{c_1}$  and  $\Delta_{c_2}$ . They contain the conflicting delta actions and have a mutually exclusive application condition.

3. Check the preconditions of *minus*( $\Delta_1, \Delta_{c_1}$ ) and *minus*( $\Delta_2, \Delta_{c_2}$ ). If any is unfulfilled, revert the previous steps and inform the developer about the unfulfilled precondition(s) and return.
4. Apply the operations yielding delta modules  $\Delta_{m_1}$  and  $\Delta_{m_2}$  that contain only the non conflicting delta actions of  $\Delta_1$  and  $\Delta_2$ .
5. Delete the delta modules  $\Delta_1$  and  $\Delta_2$ .

*Resolve Duplicated Actions*. For resolving duplicated delta actions we assemble the operations *intersect* and *minus*:

1. Check the preconditions of *intersect*( $\Delta_1, \Delta_2$ ), *minus*( $\Delta_1, \Delta_2$ ) and *minus*( $\Delta_2, \Delta_1$ ). If any precondition is unfulfilled, the refactoring is unavailable for the given delta modules.
2. Apply the operation yielding delta modules  $\Delta_i, \Delta_{m_1}$  and  $\Delta_{m_2}$ .  $\Delta_i$  contains the common delta actions,  $\Delta_{m_1}$  and  $\Delta_{m_2}$  the non-common delta actions of  $\Delta_1$  and  $\Delta_2$ .
3. Delete the delta modules  $\Delta_1$  and  $\Delta_2$ .

*Merge Delta Modules with Equivalent Conditions*. For merging two delta modules having an equivalent application condition we assemble the operations as follows:

1. Check the precondition of *concat*( $\Delta_1, \Delta_2$ ). If it is fulfilled, apply the operation yielding the delta module  $\Delta_{con}$  and delete the delta modules  $\Delta_1$  and  $\Delta_2$ . Otherwise, check if both delta modules are in conflict. If they are in conflict inform the developer about the unfulfilled precondition and return, otherwise:
  1. Check the precondition of the refactoring *Resolve Duplicated Actions*. If its precondition is unfulfilled, inform the developer and return, otherwise apply the refactoring.
  2. Check the precondition of *concat*( $\Delta_i, \Delta_{m_1}, \Delta_{m_2}$ ), where  $\Delta_i, \Delta_{m_1}$  and  $\Delta_{m_2}$  are the delta modules resulting from the previously applied refactoring. Again, if any precondition is unfulfilled, revert the previous steps, inform the developer and return. Otherwise, apply the operation yielding the delta module  $\Delta_{con}$  and delete the intermediate delta modules  $\Delta_i, \Delta_{m_1}$  and  $\Delta_{m_2}$ .
3. If the precondition of *purge*( $\Delta_{con}$ ) is fulfilled, i.e.,  $\Delta_{con}$  contains transient effects, apply the operation.

*Remove Dead Delta Action*. To remove a dead delta action, we assemble our operations as follows.

1. Check the preconditions of *concat*( $\Delta_1, \Delta_2$ ) and inform the developer if there are conflicting delta actions. Otherwise, if there are duplicated delta actions, try to eliminate them analogously to the refactoring *Resolve Duplicated Delta Actions*, concatenate the resulting delta actions to  $\Delta_{con} = concat(intersect(\Delta_1, \Delta_2), minus(\Delta_1, \Delta_2), minus(\Delta_2, \Delta_1))$  and delete the intermediate results of *intersect* and *minus*. If there are no duplicated delta actions, call  $\Delta_{con} = concat(\Delta_1, \Delta_2)$ .

2. Eliminate the transient effects in  $\Delta_{con}$  by calling  $purge(\Delta_{con})$ .
3. Check the precondition of  $\text{intersect}(\Delta_j, \Delta_{con})$  with  $1 \leq j \leq 2$ . If the preconditions are fulfilled, apply  $\Delta_{i,j} = \text{intersect}(\Delta_j, \Delta_{con})$ . Otherwise, revert the previous steps, inform the developer about the unfulfilled preconditions and return.
4. Delete the delta modules  $\Delta_1$ ,  $\Delta_2$  and  $\Delta_{con}$ .

## 6 CASE STUDY

We performed a case study by implementing the PPU scenario as presented, in part, by our running example in order to demonstrate suitability and applicability of our approach. In the following, we report how several problems that arose were detected by our analysis functions and fixed by applying appropriate refactorings. Applicable refactorings are determined by the SiPL tool environment, while the actual selection is left to the discretion of the developer. More generally, this interactive refactoring process can be guided by calculating the impact of refactorings on a set of quality metrics [21], which is out of the scope of this paper.

We chose the minimal core model CM comprising 258 model elements, a subset of which was introduced in Section 2. For each concrete feature having no cross-tree constraints, we created a new delta module. Then, we used our analysis facilities to detect (problematic) delta module interrelations and, if available, we applied refactorings to solve them. Otherwise, we inspected the interrelations and solved them manually. Finally, we created a delta module for each feature interaction and repeated the previous step until all problematic interrelations were resolved<sup>2</sup>.

We started with the separate implementation of the features SSORTATION and DSORTATION. Table 3 presents elementary information on the delta modules (DM), along with their application conditions ( $\varphi$ ) and the amount of contained delta actions (DA). The other columns show the amount of detected (problematic) interrelations which involve the delta module, i.e., dependencies (DEP), conflicts (CON), duplicates (DUP) and transient effects (TE). For each delta module in a table row, superscript letters attached to these numbers indicate the other delta modules which are involved in interrelations. In Table 3, we have  $a = \Delta_{DSort}$  and  $b = \Delta_{SSort}$  (see table caption), which means that all the 41 duplicate delta actions of  $\Delta_{SSort}$  are shared with  $\Delta_{DSort}$  and vice versa. The high amount of duplicate delta actions is due to the fact that we started with independently implementing concrete (alternative) features to avoid undesired dependencies between the respective delta modules.

DM	$\varphi$	DA	DEP	CON	DUP	TE
$\Delta_{SSort}$	SSORTATION	53	-	-	41 <sup>a</sup>	-
$\Delta_{DSort}$	DSORTATION	74	-	-	41 <sup>b</sup>	-

**Table 3: Snapshot 1 of the PPU implementation. Superscript letters attached to interrelations indicate the following delta modules: a =  $\Delta_{DSort}$ ; b =  $\Delta_{SSort}$**

The revealed 41 duplicate delta actions are not problematic because the delta modules are never applied together. However, as mentioned before, such duplicates could increase the maintenance effort and, thus, should be eliminated. In such a case, the precondition of the refactoring *Resolve Duplicated Actions* was fulfilled and,

<sup>2</sup>The subfeatures of the feature WORKPIECE do not lead to any variability in the implementation. They are dynamically bound at run time using guards and are therefore not taken into account.

thus, could be applied yielding the new delta module  $\Delta_{Conv}$  and the revised delta modules  $\Delta_{SSort}$  and  $\Delta_{DSort}$ , which are shown in Table 4. The application condition of  $\Delta_{Conv}$  was SSORTATION  $\vee$  DSORTATION but we revised it to the equivalent condition CONVEYOR. The delta modules  $\Delta_{SSort}$  and  $\Delta_{DSort}$  have several dependencies to the newly created one and, thus, must be applied after it, which is compliant with the feature dependencies specified in the FM and, thus, is not problematic.

Next, we implemented the concrete features APRESSURE and SPRESSURE<sup>3</sup>. The respective delta modules  $\Delta_{APres}$  and  $\Delta_{SPres}$  as well as their interrelations are summarized in Table 4.

DM	$\varphi$	DA	DEP	CON	DUP	TE
$\Delta_{Conv}$	CONVEYOR	41	-	1 <sup>b</sup> , 1 <sup>c</sup>	1 <sup>b</sup> , 1 <sup>c</sup>	-
$\Delta_{SSort}$	SSORTATION	12	11 <sup>a</sup>	-	-	-
$\Delta_{DSort}$	DSORTATION	33	21 <sup>a</sup>	-	-	-
$\Delta_{SPres}$	SPRESSURE	146	-	1 <sup>a</sup>	139 <sup>c</sup> , 1 <sup>a</sup>	-
$\Delta_{APres}$	APPRESSURE	167	-	1 <sup>a</sup>	139 <sup>b</sup> , 1 <sup>a</sup>	-

**Table 4: Snapshot 2 of the PPU implementation with a =  $\Delta_{Conv}$ ; b =  $\Delta_{SPres}$ ; c =  $\Delta_{APres}$**

However, the newly created delta modules are not compliant with the feature model due to the conflict and duplicate interrelations between  $\Delta_{APres}$  and  $\Delta_{SPres}$  with  $\Delta_{Conv}$ . Furthermore, the delta modules  $\Delta_{APres}$  and  $\Delta_{SPres}$  have several delta actions in common so that multiple refactorings were applicable. For instance, we could apply the refactoring *Resolve Duplicated Actions* for eliminating the duplicates between  $\Delta_{APres}$  and  $\Delta_{SPres}$ , or between  $\Delta_{APres}$  or  $\Delta_{SPres}$  and  $\Delta_{Conv}$ . Alternatively, we could also resolve the conflicts of  $\Delta_{APres}$  or  $\Delta_{SPres}$  with  $\Delta_{Conv}$  using the refactoring *Extract Conflicting Actions*. Due to the large number of duplicate delta actions between  $\Delta_{APres}$  and  $\Delta_{SPres}$ , we decided to apply the refactoring *Resolve Duplicated Actions* first.

As positive side effect, the conflicting and duplicate delta actions of  $\Delta_{APres}$  and  $\Delta_{SPres}$  with  $\Delta_{Conv}$  were extracted to the newly created delta module  $\Delta_{St}$ . Again, we could decide between the two mentioned refactorings to resolve the duplicates or conflicts. We decided to resolve all duplicate delta actions first. Afterwards, we resolved the conflict by extracting the corresponding delta actions from each delta module leading to the two new delta modules  $\Delta_{ConvNotSt}$  and  $\Delta_{StNotConv}$  with several new dependencies. Now, the interrelations of the delta modules were compliant with the FM, i.e., all problematic interrelations were resolved and we continued with delta module  $\Delta_{i(SSort, SPres)}$  realizing the feature interaction of SSORTATION and SPRESSURE (see Table 5).

The new delta module did not introduce any problematic interrelation. Duplicate delta actions were unproblematic as the involved delta modules are never applied together. Transient effects also did not lead to any dead delta action and, thus, did not need to be eliminated. However, we could further improve delta module interrelations by eliminating duplicate delta actions between  $\Delta_{StNotConv}$  and  $\Delta_{i(SSort, SPres)}$  by using again the refactoring *Resolve Duplicated Actions*. After this refactoring, one delta module was empty and we deleted it manually. Next, we implemented interactions of the features SSORTATION and APRESSURE, DSORTATION and SPRESSURE, and DSORTATOIN and APRESSURE. One main observation was

<sup>3</sup>The cross-tree constraint between LIGHT and APRESSURE can be neglected as the features regarding workpieces do not lead to any variability in the implementation.

DM	$\varphi$	DA	DEP	CON	DUP	TE
$\Delta_{ConvSt}$	CONVEYOR ∨ STAMP	1	-	-	-	-
$\Delta_{Conv}$	CONVEYOR	39	-	-	-	-
$\Delta_{ConvNotSt}$	CONVEYOR ∧ ¬ STAMP	1	1 <sup>b</sup>	1 <sup>e</sup> , 1 <sup>f</sup>	-	-
...	...	...	...	...	...	...
$\Delta_{St}$	STAMP	137	-	-	-	-
$\Delta_{StNotConv}$	STAMP ∧ ¬ CONVEYOR	1	1 <sup>a</sup> , 2 <sup>c</sup>	1 <sup>d</sup>	1 <sup>f</sup>	-
...	...	...	...	...	...	...
$\Delta_{i(SSort,SPres)}$	SSORTATION ∧ SPRES-SURE	17	3 <sup>b</sup> , 2 <sup>a</sup> , 7 <sup>c</sup>	1 <sup>d</sup>	1 <sup>e</sup>	3 <sup>b</sup> , 3 <sup>c</sup>

**Table 5: Snapshot 3 of the PPU implementation with**  
**a=** $\Delta_{ConvSt}$ ; **b=** $\Delta_{Conv}$ ; **c=** $\Delta_{St}$ ; **d=** $\Delta_{ConvNotSt}$ ; **e=** $\Delta_{StNotConv}$ ;  
**f=** $\Delta_{i(SSort,SPres)}$

that the delta module for the interaction of SSORTATION and SPRES-SURE was covered already by the delta module for the interaction of SSORTATION and APRESSURE allowing to make use of the prior refactoring. We made similar observations for the implementation of interactions of the feature DSORTATION with SPRESSURE and APPRESSURE, which we handled analogously.

In sum, our analyses enabled us to determine a multitude of interrelations of delta modules that were not coherent with the feature model and, thus, are potentially problematic. By applying refactoring operations from our construction kit, we could resolve the majority of the resulting problems to improve the quality of the overall SPL implementation. However, in some cases, we had to perform changes manually. For instance, when extracting the common delta actions of  $\Delta_{SSort}$  and  $\Delta_{DSort}$ , we changed the application condition of the newly created delta module  $\Delta_{Conv}$  to an equivalent but shorter one. Schulze et al [29] already present such a refactoring but we have not realized it yet. The detected transient effects did not lead to dead delta actions but increase the amount of interrelations between delta modules which, in turn, may increase the analysis and maintenance effort when the MBSPL becomes more complex. Damiani et al. [6] propose an algorithm for refactoring delta-oriented product lines that do not contain such effects. We plan to offer a similar refactoring to eliminate any transient effect for MBSPLs. Finally, our analysis functions do not consider all kinds of potential conflicts or dependencies. Hence, we plan to integrate advanced graph transformation concepts such as negative application conditions. Additionally, we plan to broaden our evaluation corpus using more complex scenarios.

## 7 RELATED WORK

Several partial implementations of delta-oriented SPLE exist that focus on individual aspects of construction, analysis or maintenance, such as [10, 17–19, 25, 27, 31, 34, 35, 38, 39]. The arguably most complete tool suites for delta-oriented MBSPLs, DeltaEcore [32] and SiPL [20, 21], stem from our own work. They realize essential parts of the functionality described in this paper. However, their descriptions rely on an intuitive understanding of delta operations and the effects of their actions. This work extends them by conclusive analyses and refactorings that prevent unintended side-effects.

Various analysis strategies for ensuring the correctness of delta-oriented SPLs have been proposed [36]. They typically assume that products are implemented in a programming language. Even if

modeling languages would be used, these analyses assume that the source artifacts have known semantics; they depend on these semantics. In contrast to this, our approach operates on a syntactical level and is applicable to arbitrary modeling languages.

To the best of our knowledge, only Jayaraman et al. [12] present a conflict and dependency analysis of delta modules similar to ours. It is based on critical pairs of graph transformation rules. However, in [12], each feature implementation defines one special monolithic graph transformation. In contrast, we use a standard set of edit operations to implement delta modules. Furthermore, the approach of Jayaraman et al. detects only dependencies and conflicts.

Support for the automatic refactoring of delta-based SPLs has been proposed in [29] and [6]. Schulze et al. [29] give a catalogue of refactorings and additional tool support for executing them. However, the contexts for refactorings, i.e., the problematic interrelations between delta modules, which are automatically found by our analysis functions, must be identified manually. Furthermore, the set of refactorings they provide is fixed, while our construction kit allows us to assemble refactorings for a specific use case. Damiani et al. [6] propose an algorithm for refactoring delta-oriented product lines into monotonic ones. Delta modules of a monotonic increasing (decreasing) implementation can contain only add (remove) and modify operations. However, these global refactorings assume that there are no defects in the set of delta modules as addressed by our approach.

## 8 CONCLUSION

This paper presented a new approach for analyzing and refactoring sets of interrelated delta modules in delta-oriented model-based software product lines. The most important difference to other work is that we formulate delta modules and delta actions based on graph transformation concepts, which provides a formal specification of their effects. This enables us to implement (i) analysis functions that offer detailed insights into the relations between delta modules and (ii) various delta-generating operations, which can be used to realize refactorings. We demonstrated suitability and applicability of these facilities within a case study from the automation domain. Although there is yet no empirical evidence, we believe that our approach has the potential to significantly support the ease of development and maintenance of delta-oriented model-based software product lines. While, in this paper, we concentrated on the formal foundations of our approach, we leave an empirical evaluation including studies with users of different levels of experience for future work. From a technical point of view, our analysis functions are based on pairs of delta modules due to the conflict definition stemming from graph transformation. Although this was not a problem within our case study, we will investigate a more general notion of a conflict in the future to support higher-level interactions of delta modules.

## ACKNOWLEDGMENT

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future - Managed Software Evolution.

## REFERENCES

- [1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer.
- [2] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. 2010. Henshin: advanced concepts and tools for in-place EMF model transformations. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 121–135.
- [3] Don Batory. 2005. Feature models, grammars, and propositional formulas. In *International Software Product Line Conference*. Springer, 7–20.
- [4] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. 2012. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software & Systems Modeling* 11, 2 (2012), 227–250.
- [5] Ferruccio Damiani, Reiner Hähnle, Eduard Kamburjan, and Michael Lienhardt. 2017. A Unified and Formal Programming Model for Deltas and Traits. In *International Conference on Fundamental Approaches to Software Engineering*, 424–441.
- [6] Ferruccio Damiani and Michael Lienhardt. 2016. Refactoring delta-oriented product lines to achieve monotonicity. *arXiv preprint arXiv:1604.00346* (2016).
- [7] Ferruccio Damiani, Luca Padovani, Ina Schaefer, and Christoph Seidl. 2018. A core calculus for dynamic delta-oriented programming. *Acta Inf.* 55, 4 (2018), 269–307.
- [8] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2006. *Fundamentals of Algebraic Graph Transformation*. Springer.
- [9] Martin Fowler. 1999. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley. <http://martinfowler.com/books/refactoring.html>
- [10] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Looß, Bernhard Rumpe, Klaus Müller, and Ina Schaefer. 2013. Engineering Delta Modeling Languages. In *International Software Product Line Conference*. ACM, New York, NY, USA, 22–31.
- [11] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. 2002. Confluence of typed attributed graph transformation systems. In *International Conference on Graph Transformation*. Springer, 161–176.
- [12] Praveen Jayaraman, Jon Whittle, Ahmed M Elkhodary, and Hassan Gomaa. 2007. Model composition in product lines and feature interaction detection using critical pair analysis. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 151–165.
- [13] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report, DTIC Document.
- [14] Christian Kästner, Sven Apel, Marko Rosenmüller, Don Batory, Gunter Saake, et al. 2009. On the impact of the optional feature problem: Analysis and case studies. In *International Software Product Line Conference*, 181–190.
- [15] Timo Kehrer, Gabriele Taentzer, Michaela Rindt, and Udo Kelter. 2016. Automatically deriving the specification of model editing operations from meta-models. In *International Conference on Theory and Practice of Model Transformations*. Springer, 173–188.
- [16] Leen Lambers, Daniel Strüber, Gabriele Taentzer, Kristopher Born, and Jevgenij Huetber. 2018. Multi-granular conflict and dependency analysis in software engineering based on graph transformation. In *International Conference on Software Engineering*. ACM, 716–727.
- [17] Sascha Lity, Matthias Kowal, and Ina Schaefer. 2016. Higher-order delta modeling for software product line evolution. In *International Workshop on Feature-Oriented Software Development*. ACM, 39–48.
- [18] Sascha Lity, Sophia Nahrendorf, Thomas Thüm, Christoph Seidl, and Ina Schaefer. 2018. 175% Modeling for Product-Line Evolution of Domain Artifacts. In *International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 27–34.
- [19] Michael Nieke, Gil Engel, and Christoph Seidl. 2017. DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-Aware Software Product Lines. In *International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 92–99.
- [20] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. 2015. SiPL – A Delta-Based Modeling Framework for Software Product Line Engineering. In *International Conference on Automated Software Engineering*, 852–857.
- [21] Christopher Pietsch, Dennis Reuling, Udo Kelter, and Timo Kehrer. 2017. A tool environment for quality assurance of delta-oriented model-based SPLs. In *International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 84–91.
- [22] Christopher Pietsch, Christoph Seidl, Michael Nieke, and Timo Kehrer. 2019. *Model Management and Analytics for Large Scale Systems*. Elsevier, Chapter Delta-Oriented Development of Model-Based Software Product Lines with DeltaEcore and SiPL: A Comparison. accepted.
- [23] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.
- [24] Michaela Rindt, Timo Kehrer, and Udo Kelter. 2014. Automatic Generation of Consistency-Preserving Edit Operations for MDE Tools. In *Proceedings of the Demonstrations Track of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (CEUR Workshop Proceedings)*, Vol. 1255. CEUR-WS.org.
- [25] Hamideh Sabour and Ramtin Khosravi. 2013. Delta Modeling and Model Checking of Product Families. In *Fundamentals of Software Engineering*, Farhad Arbab and Marjan Sirjani (Eds.). Springer, 51–65.
- [26] Ina Schaefer. 2010. Variability Modelling for Model-Driven Development of Software Product Lines.. In *International Workshop on Variability Modelling of Software-intensive Systems*, 85–92.
- [27] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tancarella. 2010. Delta-Oriented Programming of Software Product Lines. In *International Software Product Line Conference*, 77–91.
- [28] Ina Schaefer, Alexander Worret, and Arnd Poetzsch-Heffter. 2009. A model-based framework for automated product derivation. In *International Workshop on Model-Driven Approaches in Software Product Line Engineering*, 14–21.
- [29] Sandro Schulze, Oliver Richers, and Ina Schaefer. 2013. Refactoring Delta-oriented Software Product Lines. In *International Conference on Aspect-oriented Software Development*. ACM, 73–84.
- [30] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. 2012. Co-evolution of Models and Feature Mapping in Software Product Lines. In *Proceedings of the 16th International Software Product Line Conference - Volume 1 (SPLC '12)*. ACM, New York, NY, USA, 76–85.
- [31] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2013. Variability-aware Safety Analysis Using Delta Component Fault Diagrams. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops (SPLC '13 Workshops)*. ACM, New York, NY, USA, 2–9.
- [32] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. DeltaEcore-A Model-Based Delta Language Generation Framework.. In *Modellierung*, Vol. 19. Gesellschaft für Informatik e.V., Bonn, 21.
- [33] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. Integrated Management of Variability in Space and Time in Software Families. In *International Software Product Line Conference*. ACM, 22–31.
- [34] Christoph Seidl, Sven Schuster, and Ina Schaefer. 2015. Generative Software Product Line Development Using Variability-Aware Design Patterns. In *International Conference on Generative Programming: Concepts and Experiences*.
- [35] Christoph Seidl, Sven Schuster, and Ina Schaefer. 2017. Generative Software Product Line Development Using Variability-Aware Design Patterns. *Computer Languages, Systems & Structures* 48 (2017), 89 – 111. Special Issue on the 14th International Conference on Generative Programming: Concepts & Experiences.
- [36] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A classification and survey of analysis strategies for software product lines. *Comput. Surveys* 47, 1 (2014), 6.
- [37] Birgit Vogel-Heuser, Christoph Legat, Jens Folmer, and Stefan Feldmann. 2014. *Researching evolution in industrial plant automation: Scenarios and documentation of the pick and place unit*. Technical Report, Institute of Automation and Information Systems, Technische Universität München.
- [38] David Wille, Tobias Runge, Christoph Seidl, and Sandro Schulze. 2017. Extractive Software Product Line Engineering Using Model-Based Delta Module Generation. In *International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 36–43.
- [39] Tim Winkelmann, Jonathan Koscielny, Christoph Seidl, Sven Schuster, Ferruccio Damiani, Ina Schaefer, et al. 2016. Parametric DeltaJ 1.5: Propagating Feature Attributes Into Implementation Artifacts, Vol. 1559. CEUR-WS, 40–54.

# Covert and Phantom Features in Annotations: Do They Impact Variability Analysis?

Kai Ludwig

Harz University of Applied Sciences  
Wernigerode, Germany  
kludwig@hs-harz.de

Jacob Krüger

Otto-von-Guericke University  
Magdeburg, Germany  
jkrueger@ovgu.de

Thomas Leich

Harz University of Applied Sciences  
Wernigerode, Germany  
leich@hs-harz.de

## ABSTRACT

The annotation-based variability of the C preprocessor (CPP) has a bad reputation regarding comprehensibility and maintainability of software systems, but is widely adopted in practice. To assess the complexity of such systems' variability, several analysis techniques and metrics have been proposed in scientific communities. While most metrics seem reasonable at first glance, they do not generalize over all possible usages of C preprocessor variability that appear in practice. Consequently, some analyses may neglect the actual complexity of variability in these systems and may not properly reflect the real situation. In this paper, we investigate two types of variation points, namely *negating* and `#else` directives, to which we refer to as *corner cases*, as they are seldom explicitly considered in research. To investigate these directives, we rely on three commonly used metrics: lines of feature code, scattering degree, and tangling degree. We (1) describe how the considered directives impact these metrics, (2) unveil the resulting differences within 19 systems, and (3) propose how to address the arising issues. The results show that the corner cases appear regularly in variable feature code and can heavily change the results obtained with established metrics. We argue that we need to refine metrics and improve variability analysis techniques to provide more precise results, but we also need to reason about the meaning of corner cases and metrics.

## CCS CONCEPTS

- Software and its engineering → Preprocessors; Software product lines; Feature interaction; Maintaining software.

## KEYWORDS

Software product lines; preprocessor; variability analysis; empirical study; software metrics

### ACM Reference Format:

Kai Ludwig, Jacob Krüger, and Thomas Leich. 2019. Covert and Phantom Features in Annotations: Do They Impact Variability Analysis?. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3336294.3336296>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-7138-4/19/09...\$15.00  
<https://doi.org/10.1145/3336294.3336296>

## 1 INTRODUCTION

The C preprocessor [14, 20], for which we show an example from Linux in Listing 1, is a widely used tool to implement variability in software [3, 13, 32]. To this end, conditional directives define *variation points* in the code (e.g., `#ifndef` in Listing 1 Line 58), whereby their *feature constants* [6] (i.e., `CONFIG_PM`) are used to include or exclude code for a specific feature configuration. This *conditional compilation* [11, 20] allows to implement internal (i.e., visible to developers) and external (i.e., visible to users) variability in a configurable software system [24, 32, 39].

The pros and cons of the C preprocessor are extensively discussed in academia [9, 28, 30, 32, 43, 47]. In particular, the software-product-line [3, 39] community is concerned with understanding, analyzing, and refactoring such variability. These efforts have led to a variety of tools [22, 34] and metrics [6] that allow to perform different analyses of preprocessor-based variability. These analyses are usually based on metrics of variation point characteristics, for instance, size, scattering, and tangling.

In a recent literature review, El-Sharkawy et al. [6] summarize such metrics and aim to define them more precisely. However, the results indicate that researchers do not always measure metrics in the same way throughout all studies (e.g., compare Liebig et al. [29] and Queiroz et al. [40]). We see three issues in using current metrics to analyze software variability: (1) varying definitions, (2) different ways of measurement, and (3) limited applicability. These issues threaten the results of studies on variable source code, prevent comparisons, and may mislead discussions on variability.

The first two issues are a concern of clearly defining metrics and their application. In this paper, we are concerned with the third issue and the conceptual terms of (i) *covert* and (ii) *phantom features* (cf. Section 2) that are not at all or wrongly captured by existing metric definitions (cf. Section 3.2): To what extent do variation points that are either (i) anonymous (i.e., `#else` directives, cf. Listing 1 Line 78) or (ii) not depending on a feature presence (i.e., negated feature expressions, cf. Listing 1 Line 58) affect variability metrics? Throughout this paper, we refer to such variability as *corner cases*, as we rarely found research that directly mentioned how to address such cases. For example, Sincero et al. [46] consider all types of conditional directives (including `#else`) as propositional formulas and focus on automated constraint solving. Liebig et al. [30] focus on specific usage patterns of variable code compared to feature expressions' semantics. However, neither study elaborates on the meaning of variation points for metrics and the consequent implications for software maintenance and evolution.

To answer the posed question, we first describe and motivate corner cases and their implications in terms of variability (cf. Section 2). Furthermore, we report an empirical study of 19 open-source

**Listing 1: Covert (#else) & phantom (#ifndef) variability in linux-4.10.4/arch/powerpc/platforms/pseries/power.c.**

```

58 #ifndef CONFIG_PM
59 struct kobject *power_kobj;
60
61 static struct attribute *g[] = {
62     &auto_poweron_attr.attr,
63     NULL,
64 };
65
66 static struct attribute_group attr_group = {
67     .attrs = g,
68 };
69
70 static int __init pm_init(void)
71 {
72     power_kobj = kobject_create_and_add("power", NULL);
73     if (!power_kobj)
74         return -ENOMEM;
75     return sysfs_create_group(power_kobj, &attr_group);
76 }
77 machine_core_initcall(pseries, pm_init);
78 #else
79 static int __init apo_pm_init(void)
80 {
81     return (sysfs_create_file(power_kobj, &auto_poweron_attr.attr));
82 }
83 machine_device_initcall(pseries, apo_pm_init);
84 #endif

```

systems (cf. Section 3). In this study, we measured how often our corner cases appear in practice, how many lines of feature code they comprise, and how they affect the commonly used metrics scattering degree and tangling degree (cf. Section 4). Overall, we derived two research questions:

- RQ<sub>1</sub>** To what extent do corner cases exist in real-world systems?  
**RQ<sub>2</sub>** To what extent do corner cases affect variability metrics?

The results show that the investigated corner cases appear regularly in several systems, for instance, influencing over one million lines of feature code in Linux. Moreover, we found that the impact on metrics can be heavy, for instance, changing the scattering degree in MySQL by 25%. We provide our tooling and all measurements of our study in an open-access repository.<sup>1</sup> Finally, we propose ways to address such corner cases and how to better understand their meaning as well as their importance for variability analysis.

## 2 MOTIVATION

In this section, we first describe and motivate the problem of covert and phantom features (cf. Listing 1). We then describe the individual corner cases we are concerned with in more detail.

### 2.1 Problem Statement

It is a general problem if metrics mislead developers in their expectations of what is measured [10, 41], as we experienced ourselves. While performing our own analyses of C preprocessor variation points and investigating related studies [29, 30, 40], we searched for the most wide-spread features using scattering degree and the most complex feature interactions using tangling degree (we define these metrics for our study in Section 3.2).

However, we found that the corresponding metric definitions [6] do not generalize over all possible variation points. Basically, scattering degree and tangling degree count feature constants [6, 29, 40]. Guided by these metrics, developers may then assign code locations

<sup>1</sup><https://bitbucket.org/ldwxlnx/splc2019data.git>

1 <b>#ifdef A</b> 2 // ... 3 #else 4 // ... 5 #endif	1 <b>#ifndef A</b> 2 // ... 3 #elif ! defined(B) 4 // ... 5 #endif	1 <b>#if ! defined(A)</b> 2 // ... 3 #elif ! defined(B) 4 // ... 5 #endif
--	--	---

(a) #else.

(b) #ifndef.

(c) Negations.

**Figure 1: Examples of general corner cases we investigated.**

to respective features. For instance, regarding the feature constant CONFIG\_PM in Listing 1, the question arises, whether—although enclosed by an ifndef-block—lines 58 to 78 really do belong to this feature? They are only affected if the feature CONFIG\_PM is deselected, which still aligns to most metric definitions, but seems unreasonable due to the negating directive.

A different problem arises, as it is unclear whether other features affect this code region, for example, because of an alternative dependency that is not represented in the code [21]. Similarly, the code may be intended to be mandatory, for instance, because the feature CONFIG\_PM is optional, but requires overwriting of base code. In other words, it is undecidable at first glance, whether such variation points are just *phantom features*, meaning that their feature constant suggests an erroneous affiliation to the respective feature.

Even worse, Lines 78 to 84 are not handled by scattering degree and tangling degree at all, due to missing feature constants in the #else directive. For that reason, a potential relation to a specific feature is *covert* and requires manual or automatic inspection of its context (i.e., by analyzing all preceding directives in that particular group). In our example in Listing 1, the actual feature code is present only in these lines, demanding the selection (presence) of the feature CONFIG\_PM. This results in a blind spot for variability analyses and developers that rely on the corresponding metrics.

### 2.2 Corner Cases

Several analyses rely on metrics to infer implications from variability in code [6]. However, vague, varying, and incomplete metric definitions can result in deviations. For instance, the tangling degree of variation points simply counts the number of feature constants in a directive, arguing that the variable code is affected by the defined feature. This is reasonable for the usual #ifdef, #if, and #elif directives, but not for the corner cases that we consider in the following. We display corresponding examples in Figure 1.

**#else Directives.** Considering already simple #else directives, as in Figure 1a, the question arises, whether feature A impacts the directly following code (i.e., Line 4)? We could argue that the code depends on the absence of feature A, and thus is closely related to it. However, from a maintenance perspective, this argumentation poses problems, for instance, supposing the task: “Perform a walkthrough of all code sections contributing to feature A.” Would we inspect only the code following the #ifdef directive, only the code following the #else directive, or both?

Because #else directives induce *covert* variability, they are obstacles for scattering degree, tangling degree, or any other metric solely relying on feature constants. As aforementioned, #else directives may represent various kinds of variability, such as:

- Code that is solely related to the absence of feature A.
- Base code, if feature A needs to override it.
- Code that is related to other features, for example, if feature B is in an alternative dependency to feature A.

Considering the third case, metrics may provide biased results not only for feature A, but also for feature B that is not accounted for.

*Negating Directives.* In Figure 1b and Figure 1c, we display cases to which we refer to as negations. For example, an `#ifndef` directive (cf. Figure 1b) is the inverse of an `#ifdef` directive (cf. Figure 1a). Many metrics still account its feature constant and thereby the code it encloses to feature A. Moreover, the code in the `#else` directive is only included if feature A is selected, which renders the `#ifndef` directive a *phantom* feature location.

In more complex situations, such as in Figure 1c, the problems become more challenging. The feature constants in each directive are negations, meaning that the corresponding code is only selected in more specific situations than some metrics would indicate. For example, simply counting the number of feature constants results in a tangling degree of one for each directive. However, when developers use such metrics as indicators to infer the complexity of feature interactions—which was probably intended by the inventors [29]—an observation of one directive alone is misleading. Since the `#elif` directive actually requires feature A to be present and B to be absent, there is a relation between two features, which may imply a tangling degree of two. Further considering `#elif` directives, the dependencies can become complex, while the preprocessor needs to check only a single feature in each directive. Again, the issue arises how different metrics may lead to irritating measurements depending on the structure of the code. Eventually, scattering degree and tangling degree count object macro names in source code—no more, no less. This does only partly allow for semantic conclusions based on the metrics’ respective values.

*Addressing Corner Cases.* Properly interpreting all corner cases requires domain knowledge or technical solutions, such as SAT or CSP solvers [46, 48]. Still, if our cases are really corner cases that rarely appear, they may be negligible. In the remaining paper, we investigate this issue to improve the awareness for such cases and propose some initial solutions to mitigate them. We do not claim that our solution is ideal or that existing metrics, on which we also rely, are unsuited for variability analysis. Our goal is to raise awareness for such problems, aiming to initiate further research on their importance and solutions.

### 3 STUDY DESIGN

In this section, we report the details of our study design, namely our *subject systems*, *metrics*, *methodology*, and *tooling*, which other researchers can reuse to reproduce our study.

#### 3.1 Subject Systems

To answer our research questions, we aimed to investigate a set of real-world and differently sized software systems that comprise preprocessor variability. For this purpose, we selected an initial set of 20 popular and still maintained open-source systems from previous works [29, 30, 33, 40]. Afterwards, we tested our analysis tooling [27] in a pilot study to identify and fix potential errors.

During this phase, we found that our tool could not parse 69 files from the test suite of the GNU Compiler Collection (GCC), which comprises syntactically malformed input files for testing the fault and recovery behavior of the C preprocessor and compiler. Consequently, we omitted the GCC in this study and analyzed 19 systems.

Within Table 1, we show our subject systems, which cover a variety of domains (e.g., web servers, operating systems), development periods, and sizes. In particular, we analyzed the Linux Kernel, which is one of the most common subject systems for variability analysis, due to its size of almost 15 million source lines of code and its practical importance. The feature prefixes represent constants that are commonly used and established for external, customer-visible features. For Linux, this is the well-known `CONFIG_` prefix, while Vim uses an abbreviation of feature (`FEAT_`). Furthermore, most systems rely either on prefixes induced by the GNU Autotools suite (`HAVE_`) or make use of prefixes based on established naming conventions (`USE_`). In this study, we focus on such external features [39], while omitting internal variability that may only be used during development. We are aware that these features represent only a subset of each system’s full variability and that the selection is not perfect and may distort our results. Nevertheless, we still cover a large set of variability and the extent of corner cases in the features we inspect can be seen as a lower boundary: Only more corner cases are possible, not less. While the ratio of corner cases in the remaining code may be smaller, we still found large differences for some systems. Moreover, we argue that the situation is comparable throughout the selected systems’ variability and also for other systems—considering that we relied on established and still maintained open-source projects, which are similar to industrial systems in terms of C preprocessor usage [13].

#### 3.2 Metrics

We aimed to understand the impact of `#else` (covert features) and negating (phantom features) directives on variability analysis. To this end, we discuss respective implications qualitatively and provide a quantitative analysis based on the following three metrics:

LoF *Lines of Feature Code* counts the number of lines that are enclosed by a conditional directive, without excluding whitespaces or comments, as defined by Liebig et al. [29]. Regarding the feature `CONFIG_PM` in our Linux example (cf. Listing 1), the area between the `#ifndef` (line 58) and the `#else` directives (line 78) comprises 19 lines of feature code.

SD *Scattering Degree of Variation Points* measures how many variation points are affected by a specific feature constant. For instance, if `CONFIG_PM` in Listing 1 would not appear in any other part of the system’s code, the scattering degree for this feature would be one.

TD *Tangling Degree of Variation Points* represents the counterpart to the scattering degree. It measures the number of different feature constants in a single variation point (directive), for example, for the `#ifndef` in Line 58 of Listing 1 the tangling degree is also one.

We strictly follow these revised definitions of scattering degree and tangling degree of Queiroz et al. [40], for which feature constants of enclosing conditional directives are **not** added to currently investigated expressions, as, for instance, Liebig et al. [29] do.

**Table 1: Overview of the subject systems that we considered for this study: The version we used, each version’s release year, the domain, development start, and size of C code. We further show the feature prefixes we investigated with the corresponding number of analyzed ( $N_{\text{Feat}}$ ) and the total number of feature expressions ( $N_{\text{Total}}$ ).**

System	Version	Year	Domain	Since	#SLOC (C)	Feature Prefixes	$N_{\text{Feat}}$	$N_{\text{Total}}$
APACHE	8.1	2017	Web server	1995	153,357	(HAVE USE)_	86	1,000
COPYTHON	3.7.1rc1	2018	Program interpreter	1989	426,942	(HAVE USE)_	686	4,295
EMACS	26.1	2018	Text editor	1985	330,196	(HAVE USE)_	680	2,327
GIMP	2.9.8	2018	Image editor	1996	761,314	(HAVE USE)_	90	1,996
GIT	2.19.0	2018	Version control system	2005	206,239	(HAVE USE)_	65	821
GLIBC	2.9	2018	Programming library	1987	818,176	(HAVE USE)_	409	5,217
IMAGEMAGICK	7.0.8-12	2018	Programming library	1987	342,797	(HAVE USE)_	5	993
LIBXML2	2.7.2	2018	Programming library	1999	169,761	(HAVE USE)_	117	2,360
LIGHTTPD	1.4.50	2018	Web server	2003	49,693	(HAVE USE)_	173	450
LINUX KERNEL	4.10.4	2017	Operating system	1991	14,746,931	CONFIG_	11,011	36,082
MYSQL	8.0.12	2018	Database system	1995	153,157	(HAVE USE)_	355	4,901
OPENLDAP	2.4.46	2018	Network service	1998	287,066	(HAVE USE)_	347	1,377
PHP	7.3.0rc2	2018	Program interpreter	1985	894,426	(HAVE USE)_	1,162	5,977
POSTGRESQL	10.1	2017	Database system	1995	790,282	(HAVE USE)_	387	2,585
SENDMAIL	8.12.11	2018	E-mail server	1983	85,639	(HAVE USE)_	24	1,223
SUBVERSION	1.10.2	2018	Version control system	2000	967,225	(HAVE USE)_	39	1,008
SYLPHED	3.6.0	2018	E-mail client	2000	117,980	(HAVE USE)_	75	417
VIM	8.1	2018	Text editor	2000	343,228	(HAVE USE FEAT)_	1,378	2,570
XFIG	3.2.7a	2018	Graphics editor	1985	109,341	(HAVE USE)_	29	193

### 3.3 Methodology

We addressed our research questions as follows: For **RQ<sub>1</sub>**, we analyzed to what extent corner cases exist within our subject systems based on quantitative data. To address **RQ<sub>2</sub>** and to demonstrate the impact of corner cases on metrics, we used our tooling to infer all feature constants logically involved in `#else` directives (cf. Section 4.4). This allows us to compare scattering degree and tangling degree measurements with and without inspecting `#else` directives. We use the differences as impact indicator and discuss the measurements qualitatively.

**RQ<sub>1</sub>: Existence of Corner Cases.** To investigate to what extent our subject systems are affected by corner cases, we analyzed the cases’ impact on variability based on two aspects:

- **Frequency:** We counted all occurrences of conditional directives that comprise the feature constants under inspection (cf. feature prefixes in Table 1). Furthermore, we grouped these occurrences according to the respective keyword (i.e., `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`) to obtain a detailed overview of the types of variability.
- **Size:** For each occurrence that we found, we additionally computed the respective lines of feature code. Again, we subdivided the results into the respective groups of keywords. By doing so, we investigated what textual volume corner cases contribute to each system.

The results demonstrate the spreading and volume of the investigated variation points within the subject systems, allowing us to reason about their proportions and impact. We examined all conditional directives that applied to the same filtering conditions,

namely that at least one feature constant matches the feature prefixes that we show in Table 1 (i.e., `CONFIG_`, `FEAT_`, `HAVE_`, `USE_`). We then analyzed the identified variability as follows:

**Totality.** We counted the occurrences and lines of feature code grouped by the directives’ keyword. Thus, we summarized the overall amount and size of variability under inspection.

**Absence.** We counted the occurrences and lines of feature code of simple negations for the keywords `#if` and `#elif`. Such negations consist of one unary logical negation (!), an optional defined operator, and one feature constant (e.g., `CONFIG_PM`). We define this subset of directives as *simple negations*, because their meaning can be easily interpreted by manual inspection. In contrast, we omitted complex expressions (e.g., `!(A && B) || C`). We did this, because the categorization into absence or presence conditions for the feature constants in such expressions requires to solve complex constraints in propositional or higher-order logic.

**Presence.** We separately counted the occurrences and lines of feature code for `#else` directives representing expressions that are neither simple negations (cf. Figure 1a) nor a complex expression, but imply requested feature constants. This means, that we inferred whether a presence condition is enforced by an `#else` directive and assigned this condition as feature expression (cf. Figure 1b). Logically, such directives’ expressions consist of a defined operator and exactly one feature constant.

**#if(n)def Directives.** As defined in the C language standard [14], the directives `#ifdef A` and `#ifndef A` are semantically enriched keyword equivalents of `#if defined(A)` and `#if ! defined(A)`, respectively. Since the language standard describes that a defined operator’s argument is exclusively a single macro name, this group of conditional directives always represents simple presence or

### Listing 2: Revealing hidden SD and TD values.

```

1 #if defined(A) && defined(B)
2 // interaction (presence) of A and B
3 #elif defined(C)
4 // simple presence of C (and absence of A and B)
5 #else
6 // interaction (absence) of A, B, and C
7 #endif

```

absence conditions. We counted these classes of directives unrestricted, aside from the precondition that the feature constants matches a defined prefix.

**RQ2: Corner Cases' Impact on SD and TD.** The scattering degree and tangling degree relate to feature constants and are applied at the level of individual conditional directives. So, their conventional application [6, 29, 30, 40] disregards `#else` directives, due to non-existing feature expressions and constants. We demonstrate the impact of `#else` directives on scattering degree and tangling degree by pointing out the amount of ignored variation points.

To this end, we measured scattering degree and tangling degree with and without the inspection of `#else` directives. As both metrics measure at the level of variation points, differences here are problematic to visualize. For this reason, we use a condensed overview by summing up scattering degree and tangling degree values with and without inspecting `#else` directives for each of our subject systems. With respect to Listing 2, this results in a scattering degree of three for the feature constants A, B, and C, as each exists only at one location. The directive in Line 1 has a tangling degree of two and the directive in Line 3 of one, summing up to three, too. In this example, we completely ignored the meaning of `#else` directives, strictly following the aforementioned metric definitions.

Then, we applied the metrics to all `#else` directives, namely on their inferred feature expressions. For our example in Listing 2, the `#else` directive in Line 5 represents the expression:

```
!((defined(A) && defined(B)) || defined(C))
```

Now, every feature constant exists twice, increasing the scattering degree to two for each feature constant and summing up to six. The tangling degree remains the same as before for Lines 1 and 3, but the `#else` directive has a tangling degree of three, due to the three different feature constants within the expression it represents, resulting in an overall value of six, too.

By computing the differences for each system, we investigated the impact of `#else` directives for both variability metrics. Regarding our examples, the differences for scattering degree and tangling degree are three, for each. Thus, `#else` directives affect both metrics by 50%, considering our example in Listing 2.

### 3.4 Tooling

In order to analyze our subject systems, we continued to implement our Java application FeatureCoPP [26].<sup>2</sup> FeatureCoPP searches a specified folder for all C header (.h) and implementation (.c) files of a software system. In parallel, it analyzes the usage of conditional directives within the found files. FeatureCoPP is a purely text-based analysis tool, similar to the C preprocessor. Still, compared to TypeChef [18, 19] and SuperC [12], which actually parse the

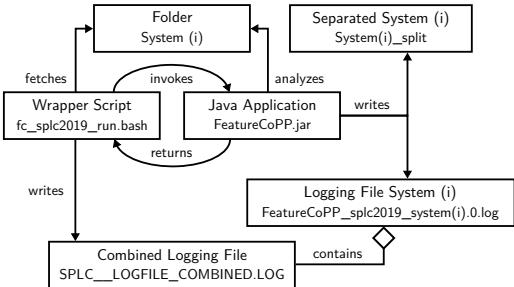


Figure 2: Conceptual behavior of our tool-chain.

code, FeatureCoPP performs equally well in identifying presence conditions [27].

We integrated FeatureCoPP into a bash-script tool-chain to automate the analysis and data collection, facilitating the replication of our study. In Figure 2, we depict the general workflow of our tool-chain. First, our *wrapper script* fetches specified systems from their repositories. Then, the script invokes *FeatureCoPP*, which analyzes the systems according to its configuration. During the analysis, FeatureCoPP creates two outputs for each system:

- (1) A new system with physically separated features and a report on these features. This is the original purpose of FeatureCoPP [26] and facilitates manual inspection of features, which we used to test our tooling and verify our data.
- (2) A *logging file* that tracks the analysis and summarizes the data we need as a statistical overview.

To facilitate reviewing of all systems (i.e., 19 logging files for this study), our wrapper script extracts and combines the statistics of all logging files into a single file. This *combined logging file* provides a condensed and more comprehensible overview of all systems and the corresponding data.

**Reproducing this Study.** We published our tooling and data in an open-access git repository<sup>1</sup> to enable other researchers to use our setup, for example, to replicate our study, to employ it on other systems, and extend it. In the repository, we provide an extended documentation on how to use and adapt our tool-chain. Moreover, we describe the details of our study and tagged the revision we used, ensuring reusability of this study's setup, despite future developments of FeatureCoPP.

## 4 RESULTS & DISCUSSION

In this section, we first discuss the suitability of our analysis for answering our research questions. Then, we present our results, discuss their implications, and propose how to address corner cases.

### 4.1 Appropriateness of Investigated Variability

**Results.** Before presenting our findings, we evaluate the appropriateness of our investigated subset of features. The question arises, whether we analyzed a representative portion of variability in our subject systems? In Table 2, we show the corresponding statistics. For example, we analyzed 278 variation points (i.e., conditional directives) in the Apache web-server that are induced by the Auto-tools (HAVE\_) or by coding conventions (USE\_). In contrast, we ignored 1,995 variation points, consisting of include guards (directives

<sup>2</sup><https://github.com/lwxlnx/FeatureCoPP>

**Table 2: Variation points in our subject systems.**

System	Total	Analyzed	%	Ignored
APACHE	2,273	278	12.23	1,995
CPYTHON	7,997	1,320	16.51	6,677
EMACS	6,524	2,701	41.40	3,823
GIMP	3,763	256	6.80	3,507
GIT	1,506	121	8.03	1,385
GLIBC	17,766	1,167	6.57	16,599
IMAGEMAGICK	3,250	6	0.18	3,244
LIBXML	9,859	427	4.33	9,432
LIGHTTPD	1,101	473	42.96	628
LINUX	102,939	49,771	48.35	53,168
MYSQL	10,328	1,237	11.98	9,091
OPENLDAP	3,992	1,008	25.25	2,984
PHP	17,837	3,474	19.48	14,363
POSTGRESQL	7,796	1,371	17.59	6,425
SENDMAIL	3,422	49	1.43	3,373
SUBVERSION	7,607	381	5.01	7,226
SYLPHED	1,670	559	33.47	1,111
VIM	15,489	10,713	69.17	4,776
XFIG	523	66	12.62	457

assuring the singular parsing of header files), directives controlling internal variability (e.g., WIN32), but also unidentified external variability. However, to identify the missing external variability, we would need specific domain knowledge about the features in each of our subject systems. In total, we examined 12.23% of all variation points in Apache.

**Discussion.** We examined a noticeable low number of variation points for ImageMagick (six out of 3,250; 0.18%) and Sendmail (49 out of 3,422; 1.43%). Arguably, these systems are outliers that require a more appropriate selection of feature prefixes (cf. Table 1) based on domain knowledge, which we aim to address in future research. Although the results for other systems also indicate relatively low percentages of analyzed variability (e.g., libxml2), we argue that we examined a reasonable amount of directives to assess the impact of our corner cases. In 12 out of 19 subject systems, we investigated more than 10 percent of the overall variability, ranging from MySQL (1,237 out of 10,328 variation points; 11.98%) to Vim (10,713 out of 15,489 variation points; 69.17%). Especially for larger systems, such as Vim, Linux, and Emacs, we analyzed high ratios of variability.

#### Insight:

Overall, we argue that our subject systems are a sound foundation for our analysis. We deliberately did not exclude outlier systems, in which we analyzed less variability, to show the complete picture and see whether the results are comparable.

## 4.2 RQ<sub>1</sub>: Existence of Corner Cases

**Results.** In Table 3, we show the number of matched directives we analyzed (N) and their total lines of feature code (LoF). We display these values grouped by directive and for each of our subject systems. Furthermore, we highlight corner-case directives with gray columns. Considering #else directives, we analogously show the

corner cases in which these enclose code that is related to a feature presence. That is, its preceding #if or #ifndef forms a simple absence condition for a particular feature constant (cf. Listing 1 and Section 3.3). At the bottom of Table 3, we summarize statistical properties of each measurement based on the actual ratios in percent to show the extent of corner cases compared to the overall variability we analyzed.

In Table 4, we summarize the measurements from Table 3. To this end, we summarize the occurrences (N) and sizes (LoF) of examined directives (All) and their respective subset of corner cases (CC) in order to show how the latter affect each subject system. We also show the percentages of corner cases within each subject system for an easier interpretation of their impact.

**Discussion: #else Directives.** We show the total number of features related #else directives per system in the second last compound column (All) in Table 3. In every system, #else directives are used in conjunction with the feature constants that we investigated—sometimes rarely, as with two occurrences in ImageMagick comprising 13 LoF, or more frequently, as with 10,449 occurrences in Linux comprising 109,750 LoF. The median and mean values of around 17% demonstrate the quantitative impact of #else directives in all systems. We can explain the relatively high dispersion ( $s = 7.17$ ) based on three factors:

- (1) *Heterogeneous sizes:* The systems differ in their sizes, which can also impact the corner cases' occurrences and sizes.
- (2) *Heterogeneous coding style:* Developers may avoid #else directives for project specific reasons (e.g., coding standards).
- (3) *Inappropriately selected feature prefixes for subject systems:* As we examined only a subset of the variability in some of our subject systems (cf. Section 4.1), the measurements of these systems may bias the overall image.

In particular, #else directives that follow after a simple absence condition (i.e., #if ! defined and #ifndef) occur rarely (e.g., one in ImageMagick, Libxml2, Xfig, and Sendmail, each). Only seven subject systems comprise such variation points in more than ten cases. Nevertheless, we argue that 488 locations comprising 13,309 LoF in the Linux Kernel or the 15 variation points with 1,340 LoF in Glibc indicate the relevance of this group of corner cases.

#### Insight:

We argue that the omnipresence of #else directives in all subject systems, the comparatively high number of such variation points, and especially their extent in the Linux Kernel underpin their significance and importance for variability analysis and management.

**Discussion: Negating Directives.** The quantitative impact of simple absence conditions related to #if and #elif directives apparently converges towards zero, with a mean of 2.54% in terms of occurrences and 2.48% in lines of feature code for #if and a median of 0% in both metrics for #elif (cf. Section 4.1). The noticeable 33.3% (#if) for ImageMagick originate from the small number of analyzed directives and resemble an outlier system. In contrast, negations appear quite regularly as #ifndef directives (i.e.,  $5.91 \pm 5.23\%$  of the code). Again, ImageMagick induces a bias on the measurements as an outlier system. We have to mention that we cannot draw

**Table 3: Overview of the analyzed preprocessor directives, including the number of variation points (N) and the corresponding lines of feature code (LoF). To investigate corner cases, we separately list the values for simple negations in #if, #ifndef, and #elif directives (Absence) as well as situations where an #else block is connected to such a negation. So, the #else directive indicates the presence of code if the previously checked feature expression is true. To make the results more comprehensible, all corner cases are highlighted in gray. Thereby, we distinguish between phantom (•) and covert (◦) features.**

System	#if				#ifdef		#ifndef		#elif				#else			
	All		• Absence		Presence		• Absence		All		• Absence		All		◦ Presence	
	N	LoF	N	LoF	N	LoF	N	LoF	N	LoF	N	LoF	N	LoF	N	LoF
APACHE	28	405	0	0	208	3,165	13	121	5	295	0	0	24	206	4	92
COPYTHON	275	5,357	15	73	727	14,372	53	753	37	369	0	0	228	1,977	10	80
EMACS	613	17,318	64	2,191	1,391	50,669	158	4,418	74	1,655	3	23	465	8,149	30	710
GIMP	60	1,242	0	0	130	2,100	22	300	14	233	0	0	30	286	6	137
GIT	21	236	1	3	65	974	8	101	2	32	0	0	25	306	3	144
GLIBC	393	5,261	14	108	416	4,196	71	905	6	57	0	0	281	3,368	15	1,340
IMAGEMAGICK	4	50	2	20	0	0	0	0	0	0	0	0	2	13	1	6
LIBXML2	47	724	3	43	263	2,238	22	115	45	465	0	0	50	577	1	7
LIGHTTPD	112	1,197	0	0	254	4,325	12	562	12	91	0	0	83	475	2	21
LINUX	7,221	161,649	118	1,454	29,805	849,807	1,409	21,055	887	6,247	11	141	10,449	109,750	488	13,309
MYSQL	97	2,248	9	761	728	7,641	74	784	43	298	0	0	295	1,727	5	24
OPENLDAP	133	2,954	1	3	571	16,304	38	827	71	857	2	10	195	1,725	5	76
PHP	1,208	98,827	20	781	1,570	25,951	125	1,013	129	1,142	2	14	442	4,329	17	1,370
POSTGRESQL	142	2,076	11	45	785	19,160	96	823	36	267	1	3	312	2,896	16	167
SENDMAIL	32	256	3	22	3	202	6	18	0	0	0	0	8	147	1	13
SUBVERSION	16	244	0	0	145	815	92	360	11	155	0	0	117	451	2	63
SYLPHED	304	7,032	0	0	189	7,246	7	28	5	42	0	0	54	317	0	0
VIM	1,870	203,255	12	124	7,598	88,503	193	3,094	25	108	0	0	1,027	11,295	32	960
XFIG	3	11	0	0	53	459	4	49	0	0	0	0	6	185	1	151
Unweighted statistical summary of ratios in percent (%)																
MEAN	24.00	28.88	2.54	2.48	49.68	51.44	5.91	4.59	2.67	2.52	0.02	0.01	17.75	12.56	1.82	3.45
MEDIAN	17.46	18.00	0.58	0.18	53.72	59.07	5.15	3.30	2.54	1.37	0.00	0.00	17.27	9.91	0.98	1.04
STD. DEV.	19.07	23.19	7.59	7.26	20.40	21.02	5.23	4.08	2.67	3.17	0.05	0.01	7.17	7.82	3.67	5.42
MIN.	4.20	1.56	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	8.63	2.16	0.00	0.00
MAX.	66.67	79.37	33.33	31.75	80.30	75.97	24.15	17.78	10.54	11.29	0.20	0.04	33.33	26.28	16.67	21.45

conclusions on the influence of #else directives representing simple absence conditions. This relates to the fact that the difference of simple presence conditions from the totality of such directives also includes more complex expressions, for example, inferred from preceding #if and multiple #elif directives.

#### Insight:

While simple absence conditions appear seldom in our subject systems, we argue that the consequent use of #ifndef directives asks for analyzing and discussing these directives.

**Summary.** While single corner cases may appear rarely, we can see (cf. Table 4) that the total ratios of affected directives and feature code vary heavily. For instance, in Sylpheed, we can see that corner cases affect only 2.35% of feature code, while they represent 10.91% of directives. More extreme, for Linux, we analyzed 49,771 directives in total, of which 24.08% are connected to our corner cases, affecting over 100 thousand (11.53%) lines of feature code.

#### Answering RQ<sub>1</sub> (existence of corner cases):

While our analysis shows that we cover only a small part of some systems, we are still able to emphasize the relevance and impact of our corner cases. The results show that these corner cases appear regularly, and thus require more attention.

### 4.3 RQ<sub>2</sub>: Corner Cases and Metrics

**Results.** In order to show the impact of our corner cases on the scattering degree and tangling degree, we show the summarized results (cf. Section 3.3) for #else directives in Table 5. For instance, within the Linux Kernel, we obtained a summarized scattering degree of 41,992 and 54,056 for omitting ( $SD_{\text{else}}$ ) and including ( $SD_{\text{else}}$ ) #else directives, respectively. This represents a noticeable difference ( $\Delta$ ) of 12,064 (22.32%). Likewise, the tangling degree is remarkably influenced in Linux (a  $TD_{\text{else}}$  of 42,335 compared to a  $TD_{\text{else}}$  of 54,634) with a loss of 22.51%, caused by not taking #else directives into account. Each of our subject systems comprises a sufficient number of #else directives, which explains average misses for the scattering degree of  $18.37 \pm 7.16\%$  and for the tangling degree of  $20.03 \pm 9.64\%$ .

**Discussion.** The dispersions for both metrics are caused by systems like Vim (9.39%, 8.97%) or Apache (8.56%, 9.27%) that comprise rather small differences. This situation may occur due to two reasons:

- (1) A modest usage of #else directives.
- (2) Less complex preceding conditional expressions, resulting in less complex expression equivalents for #else directives.

However, our results indicate that metrics that address only feature constants—and thus ignore #else directives—are not able to draw reliable conclusions on a systems' variability. We underpin

**Table 4: Comparison of corner cases (CC) to all variation points analyzed in numbers (N), size (LoF), and percent (%).**

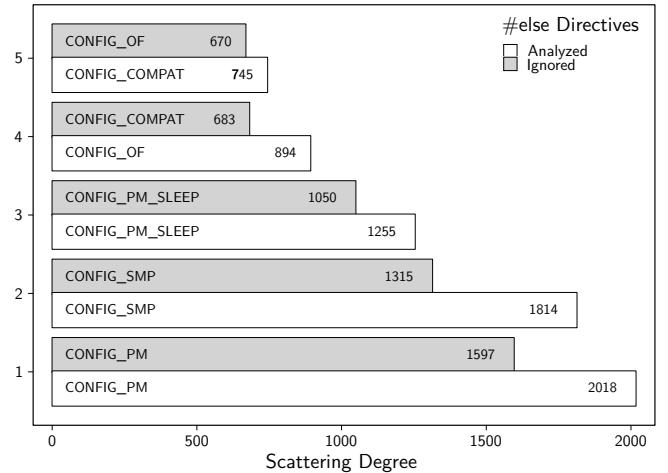
System	N			LoF		
	All	CC	%	All	CC	%
APACHE	278	37	13.31	4,192	327	7.8
CYPTHON	1,320	296	22.42	22,828	2,803	12.28
EMACS	2,701	690	25.55	82,209	14,781	17.98
GIMP	256	52	20.31	4,161	586	14.08
GIT	121	34	28.1	1,649	410	24.86
GLIBC	1,167	366	31.36	13,787	4,381	31.78
IMAGEMAGICK	6	4	66.67	63	33	52.38
LIBXML2	427	75	17.56	4,119	735	17.84
LIGHTTPD	473	95	20.08	6,650	1,037	15.59
LINUX	49,771	11,987	24.08	1,148,508	132,400	11.53
MYSQL	1,237	378	30.56	12,698	3,272	25.77
OPENLDAP	1,008	236	23.41	22,667	2,565	11.32
PHP	3,474	589	16.95	131,262	6,137	4.68
POSTGRESQL	1,371	420	30.63	25,222	3,767	14.94
SENDMAIL	49	17	34.69	623	187	30.02
SUBVERSION	381	209	54.86	2,025	811	40.05
SYLPHED	559	61	10.91	14,665	345	2.35
VIM	10,713	1,232	11.5	306,255	14,513	4.74
XFIG	66	10	15.15	704	234	33.24

our argumentation with Figure 3, in which we show an example from the Linux Kernel. In this example, we depict the five most scattered feature constants, which are ranked by their corresponding scattering degree. Furthermore, to derive this ranking, we measured with (□) and without (□) analyzing #else directives. Unsurprisingly, the scattering degree values increase for each feature constant that occurs in any #if, #ifdef, #ifndef, and #elif directive that precedes an #else directive, if we also analyzed that respective #else directive. For example, the scattering degree of feature CONFIG\_PM increases from 1,597 to 2,018, which is caused by 421 occurrences in #else directives. Remarkable is the exchange of features CONFIG\_COMPAT and CONFIG\_OF between rank four and five. Without the analysis of #else directives, the feature CONFIG\_COMPAT has a slightly higher scattering degree of 683. When we analyzed #else directives, the feature CONFIG\_OF is suddenly scattered more often with a scattering degree of 894.

Similar situations appear for the tangling degree in #else directives. For instance, for the Linux Kernel, we found a maximum tangling degree of 12 for the #if directive in Line 1202 of the file linux-4.10.4/drivers/tty/vt/keyboard.c when we did not analyze the corresponding #else directive. In contrast, we identified the feature expression with the highest number of different feature constants for the #else in Line 141 of the file linux-4.10.4/arch/mips/include/asm/module.h, which has a tangling degree of 28. Admittedly, this particular variation point comprises only a single line of feature code with no functional impact (i.e., it is an #error directive).

#### Answering RQ<sub>2</sub> (corner cases' impact on metrics):

The example and our results are good indicators for the impact of ignoring #else directives in practice: The measurements based on existing metrics may be wrong and can lead to faulty assumptions about a project's variability.

**Figure 3: Linux' five most scattered features with and without analyzing #else directives.**

#### 4.4 Tackling Corner Cases

As we demonstrated the relevance of #else and negating directives for our subject systems, we now suggest an initial idea on how to cope with the corner cases discussed.

**#else Directives.** Variability analysis tools using scattering degree and tangling degree in a blackbox approach (cf. Section 3.2) would be more precise if #else directives are also analyzed. Yet, the expressiveness of the obtained values remains limited. In case the tooling follows a white box approach—meaning that it presents developers a detailed overview of all directives examined together with their respective file locations—the quality of the analysis increases alongside with developers' variability knowledge of the system under inspection. Still, the question remains: How to obtain involved feature constants especially for bare #else directives?

As a starting point, we propose our technique that we used in our tooling (cf. Section 3.4) to perform this study. We sketch our technique conceptually in Figure 4 and display a corresponding code example in Listing 3. Our technique parses contiguous conditional directives with a generated LALR(1) parser [5], which is based on the C language standard specification [14]. During the syntactical analysis, we create an abstract syntax tree (AST), consisting of each directive's operators and operands (e.g., feature constants). Furthermore, associativity and precedence of the respective conditional directive's feature expression are preserved by the AST structure. We need to handle the directives #ifdef and #ifndef separately, as the keywords (#if) are semantically enriched with operations (i.e., !, defined). Thus, we transform such directives' ASTs into an equivalent AST containing a defined operator and, if necessary, a leading logical unary negation (!), to separate the operations from the keyword. This is important to derive correct feature expressions for potentially following #else directives. In the last step, we preserve the ordering of a complete conditional directive: All ASTs from the opening #if(def) to the closing #endif are interconnected, preserving their order in the source code. If our technique recognizes an #else directive, it interconnects all previously built ASTs with logical disjunctions (||). Finally, the

**Table 5: Comparison of unweighted and summarized scattering and tangling degrees with and without ( $\neg$ ) #else analysis.**

System	$\Sigma$		Missed		$\Sigma$		Missed	
	$SD_{\neg else}$	$SD_{else}$	%	$\Delta$	$TD_{\neg else}$	$TD_{else}$	%	$\Delta$
APACHE	267	292	8.56	25	274	302	9.27	28
CYPTHON	1,189	1,443	17.60	254	1,308	1,598	18.15	290
EMACS-26.1	2,551	3,081	17.20	530	2,692	3,268	17.63	576
GIMP	228	263	13.31	35	236	277	14.80	41
GIT	98	124	20.97	26	116	147	21.09	31
GLIBC	935	1,229	23.92	294	1,192	1,584	24.75	392
IMAGEMAGICK	4	6	33.33	2	4	8	50.00	4
LIBXML2	384	447	14.09	63	410	496	17.34	86
LIGHTTPD	453	551	17.79	98	469	569	17.57	100
LINUX	41,992	54,056	22.32	12,064	42,335	54,634	22.51	12,299
MYSQL	969	1,295	25.17	326	1,016	1,391	26.96	375
OPENLDAP	884	1,138	22.32	254	908	1,194	23.95	286
PHP	3,286	3,816	13.89	530	3,567	4,283	16.72	716
POSTGRESQL	1,104	1,442	23.44	338	1,167	1,551	24.76	384
SENDMAIL	41	49	16.33	8	52	61	14.75	9
SUBVERSION	265	382	30.63	117	278	405	31.36	127
SYLPHED	506	563	10.12	57	513	577	11.09	64
VIM	10,741	11,854	9.39	1,113	11,722	12,877	8.97	1,155
XFIG	63	69	8.70	6	61	67	8.96	6
MEAN			18.37	849.47			20.03	893.11
MEDIAN			17.60	117.00			17.63	127.00
STD.DEV.			7.16	2,729.46			9.64	2,778.26
MIN			8.56	2.00			8.96	4.00
MAX			33.33	12,064.00			50.00	12,299.00

created AST gets a logical unary negation as root node. This equivalence transformation follows De Morgan’s laws. We illustrate this procedure in Figure 4, where we use the feature expressions of three conditional directives (i.e., one `#ifdef` and two `#elifs`), to construct a semantically equivalent feature expression for the trailing `#else` directive.

Our artificial creation of expressions for `#else` directives has the following advantages:

- We can create a textual representation of an `#else` directive by traversing the AST. Afterwards, each `#else` has a virtual name, which leverages developers to spot respective code locations more easily.
- The generated AST can be applied to a SAT or CSP solver in order to test satisfiability of the `#else` directive [46, 48]. This allows to analyze `#else` directives more easily, for example, to identify dead features.

#### Insight:

The preservation of feature constants allows us to apply scattering degree and tangling degree on every variation point, which results in more precise measurements.

**Absence and Presence.** The main issue with metrics, such as scattering degree and tangling degree, is the divergence from what is actually measured and what developers may expect to infer from values obtained with such metrics [10, 41]. Since scattering degree

and tangling degree only allow to draw conclusions with regards to what feature name is somehow textually involved in what code location, they are limited to perform a sound reasoning about feature presence or absence conditions. Chances may be high that such text locations really enclose actual feature code, but, considering our results, this can barely be seen as a rule of thumb.

To improve metrics that build on counting feature constants (or other values) in preprocessor directives, the semantic evaluation of such variation points is necessary. Consequently, we need tooling that is able to perform constraint solving, since expressions in variation points do not only allow Boolean logic, but even arithmetic, bitwise, and relational operations [14]. With regards to our example in Listing 3, a metric for mapping text locations to actual features could roughly behave as follows:

*For each conditional directive in a contiguous group, repeat until #endif is reached:*

- (1) *Emulate the C preprocessor’s control flow.* In order to read the second `#elif`, the absence of feature A is necessary. Thus, a preceding expression needs to be negated and conjunctively connected to its successor expression (i.e., `!defined A && B && C`). If a directive has no predecessor and is not an `#else` directive, we can take it as is. If a directive is an `#else`, the conjunction of all negated preceding directives is exactly its feature expression.
- (2) *Find all models.* After creating a respective expression, we can introduce it to a CSP solver. For each found solution,

**Listing 3: Complex contiguous conditional directives.**

```

1 #ifdef A
2 // presence of A
3 #elif B && C
4 // presence of B and C (absence of A)
5 #elif ! defined D
6 // absence of A, B, C, and D
7 #else
8 // absence of A, B, and C, presence of D
9 #endif

```

every feature constant is counted only once if its value expands to a value different from zero in a solution model. This counting happens only once, regardless whether the same constant has a value different from zero in multiple possible solutions or not. Admittedly, this step is expensive in terms of computation time [42].

- (3) *Summarize occurrences.* After this step, for the tangling degree, we can summarize all counted feature constants, and for the scattering degree, we can increment a global counter for each constant. We arguably obtain more precise results for both metrics, as we analyze covert and phantom features.

This conceptual technique ignores the nesting of directives, though. However, it can be extended to consider surrounding directives in the same conjunctive fashion. To this end, a tool must be able to follow file inclusion directives (`#include`), since nesting structures may occur only after such inclusions.

This solution is only a rough scaffold for extending metrics that are based on feature constants towards semantic capabilities. Or simply put, this solution may be a way to let these metrics answer questions, such as: What directive does really enclose feature code?

#### Insight:

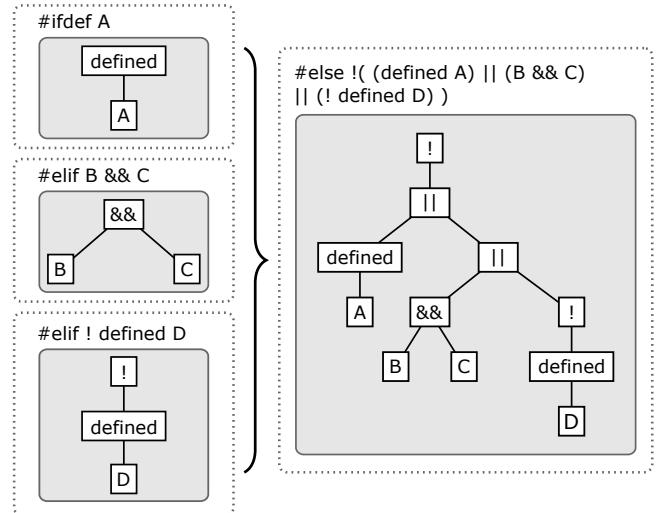
Metrics that rely on analyzing feature constants are only suitable to identify feature locations if the respective features' presence is guaranteed, and thus absence is ruled out.

## 5 THREATS TO VALIDITY

In this section, we provide an overview of threats to validity. To this end, we follow proposed classifications [38, 50] and discuss the *construct*, *internal*, and *external* validity of our study.

*Construct Validity.* Considering the construct validity, we strictly followed the metric definitions of other researchers. In particular, for lines of feature code, we used the definition of Liebig et al. [29], namely, we counted any line between consecutive conditional directives. Regarding scattering degree and tangling degree, we used the revised definitions of Queiroz et al. [40], meaning that we did not construct complex feature expressions from nested directives, as done by Liebig et al. [29]. Consequently, the results for these metrics provide insights into the impact of considering corner cases, but may not be representative about the systems themselves. They are still suitable to achieve our goal, and thus we argue that we mitigated this threat.

Moreover, we are fully aware of the shortcomings of so called "size metrics" [10, 41], such as lines of feature code. Since these only provide an absolute measure for actual **lines** of code (no matter how measured), they are never a reliable measure (and moreover language independent) to draw conclusion of the actual systems'

**Figure 4: Automatic synthesis of #else directives, exemplified for Listing 3.**

code volume. However, as these metrics are heavily used in the scientific community, we still used them to allow for comparisons to other research in this area.

*Internal Validity.* A threat to the internal validity is the fact that we implemented our own tooling, which is more lightweight compared to existing, static variability-analysis tools. There may be bugs caused by unidentified and unintended usage patterns of our tooling. However, we have tested it extensively and applied it carefully. Moreover, we already compared it to existing tools, namely TypeChef [19] and SuperC [12], and found that it performs similar or even better for our purposes [27]. Although we addressed this threat properly, we cannot completely avoid it.

*External Validity.* For the external validity, we are aware that we only consider a limited set of features, corner cases, and metrics. All these points limit our ability to generalize our results. Despite these limitations, we could already show strong impacts of corner cases on variability analysis. Considering that we found over one million feature lines of code being affected in Linux, we argue that the unveiled problems are important to consider. Furthermore, conducting our study exclusively on open-source subject systems may prevent us from generalizing the results with regards to proprietary software systems. However, Hunsen et al. [13] have demonstrated that C preprocessor usage patterns are nearly identical between open-source and closed-source systems. For this reason, we consider this limitation as tolerable and argue that our insights are also relevant for industrial systems.

## 6 RELATED WORK

Open-source systems that incorporate C preprocessor variability are common subject systems for static variability analyses, due to their availability and usage in practice. Consequently, there are numerous studies on analyzing the conditional directives of the C preprocessor. For example, Liebig et al. [29, 30] are concerned with

the discipline of preprocessor usage. They argue that using such directives in undisciplined styles, for example, annotating code below the statement level, hampers the applicability of tools and the maintainability of code. Based on these findings, Medeiros et al. [33] have proposed a set of refactorings to improve the discipline of preprocessor directives and showed that most developers prefer this refactored code. In contrast, Schulze et al. [43] report a controlled experiment in which they found no differences between disciplined and undisciplined annotations on program comprehension.

The initial discussions on problems of the C preprocessor on software development emerged from personal, negative opinions and experiences [3, 47]. To underpin these experiences, several researchers have investigated the problems of C preprocessor usage based on user studies [24, 28, 32, 45]. They partly show that the raised problems exist and can be improved, but that a lot depends on the single developer and the preprocessor usage.

Fenske et al. [9] report an empirical study on the C preprocessor's change proneness, and thus tackle the impact of negating and `#else` directives from a different perspective. To this end, the authors inspect variation points only in implementation files (`.c`) at function level. As a result, they disregard variability in interface declarations and inline function definitions [14] induced by header files (`.h`).

Sincero et al. [46] present a linear growing algorithm to create propositional formulas from C preprocessor variation points in C source code. The authors focus primarily on aspects of satisfiability to detect, for example, dead feature code. To this end, they also take `#else` directives into account and transform them into equivalent expressions, comparable to our technique (cf. Section 4.4). While the authors assumed a potential impact of their technique on the understanding of variability and corresponding metrics, they did not address this topic, as we did with this work.

Some researchers have proposed new techniques to replace or improve C preprocessor variability. Most prominently may be the concept of virtually separating concerns [15, 16] and adding background colors to highlight feature code within an integrated development environment [7, 8]. While empirical studies show advantages for both techniques, they do not seem to be adopted in practice—where conventional development tools still dominate. Other researchers have proposed to integrate, combine, or replace preprocessor code with other variability mechanisms [17, 25, 26], for example, by using projectional editing to simply switch the representations of feature code to the user [4, 35, 49].

Considering different metrics, Queiroz et al. [40] investigate whether these have special statistic properties—namely power laws. Krüger et al. [23] compare them for mandatory and optional features in Marlin. Both studies show that the metrics align to certain patterns and may help developers to better understand the code or identify critical parts. Overall, numerous tools [22, 34] and metrics [6] have been proposed to perform such analyses.

In the context of reverse variability engineering, several researchers aim to extract variability information, for example, feature constraints from preprocessor directives [36, 37] or configuration options [31]. The goal of such techniques is to understand dependencies among configuration options (and thus their features) and their relation to the source code. To support the understanding of variability in a software system and define its configuration space, the reverse engineered constraints are used to derive a variability

model of the system [1, 2, 44]. The results of our study may indicate limitations for such automated techniques, depending on how and what constraints the techniques extract and use—but can also guide new concepts to extend the currently existing techniques.

We are not aware of another study investigating corner cases of the C preprocessor and their impact on variability analysis and metrics. The aforementioned works all report and investigate other issues of the C preprocessor. Some of them also use different sets of metrics to perform their studies. Still, none of them seems to address the issue of negations or `#else` directives and their impact on metrics for variability analysis. Therefore, our work differs from existing works and can be seen as a complement that may ask for reshaping some metrics or investigating to what extent these are suitable to achieve the goals of previous studies.

## 7 CONCLUSION

In this paper, we analyzed how specific corner cases of C preprocessor directives, namely negating and `#else` directives, affect the variability analysis of such systems. To this end, we focused on determining the extent to which such cases appear in the source code of 19 open-source systems. We compared the results for scattering degree and tangling degree after applying them to our corner cases. Finally, we proposed first steps towards improving variability analysis and the conceptual understanding of corner cases. However, our most important results indicate that:

- Corner cases can hide variability information that only exist as domain knowledge.
- Corner cases appear frequently in several systems, whether as `#else` directives—inducing anonymous, *covert* variation points—or as negated feature expressions, which require the absence of a feature—yielding *phantom* variation points regarding their involved feature constants.
- Resolving corner cases can significantly change the results of variability metrics that focus on counting feature constants, such as scattering degree and tangling degree.
- Variability enclosed in corner cases has a relevant textual size, which highlights the importance of considering covert and phantom features in real-world systems.

We hope to motivate discussions and further analyses regarding evaluation, precision, and purpose of existing and upcoming variability metrics. Based on our findings, we argue that more research about such corner cases is necessary to better understand their characteristics and provide guidance for practitioners.

To this end, we plan to extend our analysis significantly, including more advanced parsers and metrics, as well as more subject systems. A particularly interesting factor that we want to focus on is the question how and why corner cases are applied: When and for what purpose are developers using them? For this purpose, we intend to conduct interview studies and surveys on various systems to collect real-world experiences and practices. This knowledge may help us to improve and scope variability analysis and to reason about the importance of considering corner cases.

## ACKNOWLEDGMENTS

This research has been supported by the German Research Foundation (DFG) project EXPLANT grants LE 3382/2-1 and LE 3382/2-3.

## REFERENCES

- [1] Mathieu Acher, Benoit Baudry, Patrick Heymans, Anthony Cleve, and Jean-Luc Hainaut. 2013. Support for Reverse Engineering and Maintaining Feature Models. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 20:1–20:8. <https://doi.org/10.1145/2430502.2430530>
- [2] Nele Andersen, Krzysztof Czarnecki, Steven She, and Andrzej Wąsowski. 2012. Efficient Synthesis of Feature Models. In *International Software Product Line Conference (SPLC)*. ACM, 106–115. <https://doi.org/10.1145/2362536.2362553>
- [3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-Oriented Software Product Lines*. Springer. <https://doi.org/10.1007/978-3-642-37521-7>
- [4] Benjamin Behringer, Jochen Palz, and Thorsten Berger. 2017. PEOPL: Projectional Editing of Product Lines. In *International Conference on Software Engineering (ICSE)*. IEEE, 563–574. <https://doi.org/10.1109/ICSE.2017.758>
- [5] Frank DeRemer. 1969. *Practical Translators for LR(k) Languages*. Ph.D. Dissertation. Massachusetts Institute of Technology.
- [6] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. 2019. Metrics for Analyzing Variability and Its Implementation in Software Product Lines: A Systematic Literature Review. *Information and Software Technology* 106 (2019), 1–30. <https://doi.org/10.1016/j.infsof.2018.08.015>
- [7] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. 2013. Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empirical Software Engineering* 18, 4 (2013), 699–745. <https://doi.org/10.1007/s10664-012-9208-x>
- [8] Janet Feigenspan, Michael Schulze, Maria Papendieck, Christian Kästner, Raimund Dachselt, Veit Köppen, and Mathias Frisch. 2011. Using Background Colors to Support Program Comprehension in Software Product Lines. In *Annual Conference on Evaluation & Assessment in Software Engineering (EASE)*. IET, 66–75. <https://doi.org/10.1049/ic.2011.0008>
- [9] Wolfram Fenske, Sandro Schulze, and Gunter Saake. 2017. How Preprocessor Annotations (Do Not) Affect Maintainability: A Case Study on Change-Proneness. In *International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 77–90. <https://doi.org/10.1145/3136040.3136059>
- [10] Norman Fenton and James Bieman. 2014. *Software Metrics: A Rigorous and Practical Approach*. CRC Press. <https://doi.org/10.1201/b17461>
- [11] Cristina Gacek and Michalis Anastasopoulos. 2001. Implementing Product Line Variabilities. *ACM SIGSOFT Software Engineering Notes* 26, 3 (2001), 109–117. <https://doi.org/10.1145/379377.375269>
- [12] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 323–334. <https://doi.org/10.1145/2254064.2254103>
- [13] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. 2016. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering* 21, 2 (2016), 449–482. <https://doi.org/10.1007/s10664-015-9360-1>
- [14] ISO/IEC. 2011. *Programming Languages - C*. Technical Report ISO/IEC 9899:201x. International Standards Organization.
- [15] Christian Kästner. 2010. *Virtual Separation of Concerns*. Ph.D. Dissertation. Otto-von-Guericke University.
- [16] Christian Kästner and Sven Apel. 2009. Virtual Separation of Concerns – A Second Chance for Preprocessors. *Journal of Object Technology* 8, 6 (2009), 59–78.
- [17] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2009. A Model of Refactoring Physically and Virtually Separated Features. *SIGPLAN Notices* 45, 2 (2009), 157–166. <https://doi.org/10.1145/1837852.1621632>
- [18] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 805–824. <https://doi.org/10.1145/2048066.2048128>
- [19] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. Type-Chef: Toward Type Checking #ifdef Variability in C. In *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 25–32. <https://doi.org/10.1145/1868688.1868693>
- [20] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language*. Prentice Hall.
- [21] Sebastian Krieter, Jacob Krüger, and Thomas Leich. 2018. Don't Worry About It: Managing Variability On-The-Fly. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 19–26. <https://doi.org/10.1145/3168365.3170426>
- [22] Christian Kröhner, Sascha El-Sharkawy, and Klaus Schmid. 2018. KernelHaven: An Open Infrastructure for Product Line Analysis. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 5–10. <https://doi.org/10.1145/3236405.3236410>
- [23] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. 2018. Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 105–112. <https://doi.org/10.1145/2517208.2517215>
- [24] Jacob Krüger, Kai Ludwig, Bernhard Zimmermann, and Thomas Leich. 2018. Physical Separation of Features: A Survey with CPP Developers. In *Symposium On Applied Computing (SAC)*. ACM, 2044–2052. <https://doi.org/10.1145/3167132.3167351>
- [25] Jacob Krüger, Marcus Pinnecke, Andy Kenner, Christopher Kruczak, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2018. Composing Annotations Without Regret? Practical Experiences Using FeatureC. *Software: Practice and Experience* 48, 3 (2018), 402–427. <https://doi.org/10.1002/spe.2525>
- [26] Jacob Krüger, Iyonne Schröter, Andy Kenner, Christopher Kruczak, and Thomas Leich. 2016. FeatureCoPP: Compositional Annotations. In *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 74–84. <https://doi.org/10.1145/3001867.3001876>
- [27] Elias Kuiter, Sebastian Krieter, Jacob Krüger, Kai Ludwig, Thomas Leich, and Gunter Saake. 2018. PClocator: A Tool Suite to Automatically Identify Configurations for Code Locations. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 284–288. <https://doi.org/10.1145/3233027.3236399>
- [28] Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 143–150. <https://doi.org/10.1109/VLHCC.2011.6070391>
- [29] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *International Conference on Software Engineering (ICSE)*. IEEE, 105–114. <https://doi.org/10.1145/1806799.1806819>
- [30] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 191–202. <https://doi.org/10.1145/1960275.1960299>
- [31] Max Lillack, Christian Kästner, and Eric Bodden. 2017. Tracking Load-Time Configuration Options. *IEEE Transactions on Software Engineering* 44, 12 (2017), 1269–1291. <https://doi.org/10.1109/TSE.2017.2756048>
- [32] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 495–518. <https://doi.org/10.4230/LIPIcs.ECOOP.2015.495>
- [33] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Transactions on Software Engineering* 44, 5 (2018), 453–469. <https://doi.org/10.1109/TSE.2017.2688333>
- [34] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, and Gunter Saake. 2014. An Overview on Analysis Tools for Software Product Lines. In *International Software Product Line Conference (SPLC)*. ACM, 94–101. <https://doi.org/10.1145/2647908.2655972>
- [35] Mukelabai Mukelabai, Benjamin Behringer, Moritz Fey, Jochen Palz, Jacob Krüger, and Thorsten Berger. 2018. Multi-View Editing of Software Product Lines with PEOPL. In *International Conference on Software Engineering (ICSE)*. ACM, 81–84. <https://doi.org/10.1145/3183440.3183499>
- [36] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining Configuration Constraints: Static Analyses and Empirical Results. In *International Conference on Software Engineering (ICSE)*. ACM, 140–151. <https://doi.org/10.1145/2568225.2568283>
- [37] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering* 41, 8 (2015), 820–841. <https://doi.org/10.1109/TSE.2015.2415793>
- [38] Dewayne E Perry, Adam A Porter, and Lawrence G Votta. 2000. Empirical Studies of Software Engineering: A Roadmap. In *International Conference on Software Engineering - Future of Software Engineering Track (ICSE)*. ACM, 345–355. <https://doi.org/10.1145/336512.336586>
- [39] Klaus Pohl, Günter Böckle, and Frank J. van Der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer. <https://doi.org/10.1007/3-540-28901-1>
- [40] Rodrigo Queiroz, Leonardo Passos, Marco Tulio Valente, Claus Hunsen, Sven Apel, and Krzysztof Czarnecki. 2017. The Shape of Feature Code: An Analysis of Twenty C-Preprocessor-Based Systems. *Software & Systems Modeling* 16, 1 (2017), 77–96. <https://doi.org/10.1007/s10270-015-0483-z>
- [41] Jarrett Rosenberg. 1997. Some Misconceptions About Lines of Code. In *International Software Metrics Symposium (METRIC)*. IEEE, 137–142. <https://doi.org/10.1109/METRIC.1997.637174>
- [42] Francesca Rossi, Peter Van Beek, and Toby Walsh. 2006. *Handbook of Constraint Programming*. Elsevier.
- [43] Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. 2013. Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment. In *International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 65–74. <https://doi.org/10.1145/2517208.2517215>

- [44] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *International Conference on Software Engineering (ICSE)*. ACM, 461–470. <https://doi.org/10.1145/1985793.1985856>
- [45] Janet Siegmund, Christian Kästner, Jörg Liebig, and Sven Apel. 2012. Comparing Program Comprehension of Physically and Virtually Separated Concerns. In *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 17–24. <https://doi.org/10.1145/2377816.2377819>
- [46] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2010. Efficient Extraction and Analysis of Preprocessor-based Variability. In *International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 33–42. <https://doi.org/10.1145/1868294.1868300>
- [47] Henry Spencer. 1992. `#ifdef Considered Harmful, or Portability Experience with C News`. In *USENIX Conference*. 185–197.
- [48] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *European Conference on Computer Systems (EuroSys)*. ACM, 47–60. <https://doi.org/10.1145/1966445.1966451>
- [49] Markus Voelter. 2010. Implementing Feature Variability for Models and Code with Projectional Language Workbenches. In *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 41–48. <https://doi.org/10.1145/1868688.1868695>
- [50] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer. <https://doi.org/10.1007/978-1-4615-4625-2>

# Identifying and Visualizing Variability in Object-Oriented Variability-Rich Systems

Xhevahire Ternava

xhevahire.ternava@lip6.fr

Sorbonne Université, UPMC, LIP6,  
Paris, France

Johann Mortara

johann.mortara@univ-cotedazur.fr

Université Côte d'Azur, CNRS, I3S,  
Sophia Antipolis, France

Philippe Collet

philippe.collet@univ-cotedazur.fr

Université Côte d'Azur, CNRS, I3S,  
Sophia Antipolis, France

## ABSTRACT

In many variability-intensive systems, variability is implemented in code units provided by a host language, such as classes or functions, which do not align well with the domain features. Annotating or creating an orthogonal decomposition of code in terms of features implies extra effort, as well as massive and cumbersome refactoring activities. In this paper, we introduce an approach for identifying and visualizing the variability implementation places within the main decomposition structure of object-oriented code assets in a single variability-rich system. First, we propose to use symmetry, as a common property of some main implementation techniques, such as inheritance or overloading, to identify uniformly these places. We study symmetry in different constructs (e.g., classes), techniques (e.g., subtyping, overloading) and design patterns (e.g., strategy, factory), and we also show how we can use such symmetries to find variation points with variants. We then report on the implementation and application of a toolchain, *symfinder*, which automatically identifies and visualizes places with symmetry. The publicly available application to several large open-source systems shows that *symfinder* can help in characterizing code bases that are variability-rich or not, as well as in discerning zones of interest w.r.t. variability.

## CCS CONCEPTS

- Software and its engineering → Software product lines; Object oriented development; Reusability.

## KEYWORDS

Identifying software variability, visualizing software variability, object-oriented variability-rich systems, tool support for understanding software variability, software product line engineering

## ACM Reference Format:

Xhevahire Ternava, Johann Mortara, and Philippe Collet. 2019. Identifying and Visualizing Variability in Object-Oriented Variability-Rich Systems. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19, September 9–13, 2019, Paris, France)*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3336294.3336311>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336311>

## 1 INTRODUCTION

Variability-intensive software systems are now the usual demand in many industry sectors. To manage their variability within a specific domain, software product line (SPL) engineering is the usual methodological process for developing them together. At the domain level, the variability of these products is commonly described in terms of their common and variable features, as reusable units, in a feature model [28]. Further, in a forward engineering approach, their features are realized in different software assets, including reusable code assets at the implementation level.

In many variability-rich software systems, which do not follow a complete SPL approach, variability is implemented with different traditional techniques, such as inheritance, parameters, overloading, or design patterns [9, 22, 56]. By these techniques, variability is implemented in code units provided by a host language, such as classes or functions, which do not align well with domain features. Therefore, occasionally and orthogonal to this main decomposition, some approaches are used for annotating (e.g., using preprocessors in C [37]) or putting into separate modules (e.g., with feature modules [6]) all lines of code that belong to each specific domain feature [7, 53]. But, while annotations in the form of conditional compilations have received significant attention, their use is often criticized for the code pollution due to `#ifdef-s` [35, 54] and for the occurrence of syntactic and semantic errors during the product derivation [31]. Feature modularization being considered as desirable, it still implies massive refactoring activities and cannot handle the fact that many variability dimensions become naturally cross-cutting concerns in code [29, 57]. Currently, it is thus acknowledged that there is still no satisfactory approach to well structure the implementation of variability in code assets [6, 42].

Our work thus takes the assumption that, in many variability-rich systems, one can keep unchanged the main decomposition of code and still be able to map the domain features to the variability implementation places in code assets. We consider that these *variability places* can be centers of attention in terms of design, with several implementation techniques used together. They can also be abstracted in terms of variation points (*vp-s*) with variants<sup>1</sup> [6], but a proper identification of the variability implementation places is then needed. There are studies on how to address variability by traditional techniques [11, 22, 46, 56], or on how to partially locate and identify domain features, mainly at the code level [8, 16, 50]. Nevertheless, there is a complete lack of approaches to identify variation points and variants [39] implemented with different techniques in a single variability-rich system. This could be due to the

<sup>1</sup>their definition is given in Section 2.1

fact that each traditional technique differently supports the implementation of *vp*-s with variants [39, 58]. Therefore, from a reverse perspective, it indicates that each *vp* requires its own way to be identified in code assets, depending on the used technique.

Herein, our contribution is threefold. First, we reuse the property of symmetry (Section 2.2), which has been previously explored in software [12, 24, 65–67]. From an interdisciplinary combination of software and civil engineering, it is used to describe some relevant and heavily used object-oriented techniques, as well as software design patterns (Section 3.1). Then, by using their property of symmetry, we propose an approach to identify the implementation of different kinds of *vp*-s and variants in a unified way (Section 3). Thirdly, we present *symfinder*, a tool support for automatic identification and visualization of the described symmetries, so that the determination of *vp*-s with variants is facilitated (Section 4).

We applied our tool approach in eight real open-source variability-rich software systems (Section 5)<sup>2</sup>. We report on the identified symmetries and their related *vp*-s and variants, showing that the toolchain, with its visualization support, helps in finding relevant patterns of implemented variability. We also gain more insight into the studied systems by using two metrics on the density and number of *vp*-s. Finally, we discuss threats to validity, limitations (Section 6), related work (Section 7), and conclude the paper by evoking future work (Section 8).

## 2 BACKGROUND

## 2.1 Variability in reusable code assets

Let us consider an illustrative example with a Java implementation of a family of geometric shapes, such as rectangles and circles (*cf.* Listing 1). What is common from `Rectangle` and `Circle` is factorized into the abstract class `Shape` using inheritance as a variability implementation technique. Besides, overloading is used to implement the two ways for drawing the shapes, namely the `draw()` method in `Rectangle`, lines 17–20 and 21–24. Despite its small size, we consider this example as representative of reusable code assets in which several techniques are used together, such as inheritance, overloading, or design patterns.

Regardless of the programming paradigm (e.g., object-oriented or functional), these reusable code assets consist of three parts: core, commonalities, and variabilities [11]. The core part is what remains of the system in the absence of any particular feature, namely the assets that are included in any software product within an SPL [61]. Commonality is a common part of the related variant parts, which are used to distinguish the software products within an SPL. After the commonality is factorized from the variability and implemented, it becomes part of the core [61], except when it represents some optional variability [59]. Such commonalities and variabilities are usually abstracted in terms of variation points (*vp-s*) with *variants*, respectively, which are related to concrete elements in reusable code assets.

By definition, a variation point identifies one or more locations at which the variation will occur, while the way that a variation point is going to vary is expressed by its variants [27]. In Listing 1,

```
1  /* Class level variation point, vp_Shape */
2  public abstract class Shape {
3  public abstract double area();
4  public abstract double perimeter(); /*...*/
5 }
```

```
/* First variant, v_Rectangle, of vp_Shape */
public class Rectangle extends Shape {
    private final double width, length;
    // Constructor omitted
    public double area() {
        return width * length;
    }
    public double perimeter() {
        return 2 * (width + length);
    }
    /* Method level variation point, vp_Draw */
    /* First variant of vp_Draw */
    public void draw(int x, int y) {
        // rectangle at (x, y, width, length)
    }
    /* Second variant of vp_Draw */
    public void draw(Point p) { // Point defined
        // rectangle at (p.x, p.y, width, length)
    }
}
```

```
26 /* Second variant, v_Rectangle, of vp_Shape */
27 public class Circle extends Shape {
28     private final double radius;
29     // Constructor omitted
30     public double area() {
31         return Math.PI * Math.pow(radius, 2);
32     }
33     public double perimeter() {
34         return 2 * Math.PI * radius;
35     }
36 }
```

**Listing 1: Example of variability implementations. The highlighted class and methods with a yellow color represent two *vp-s*, at the class and method level, respectively**

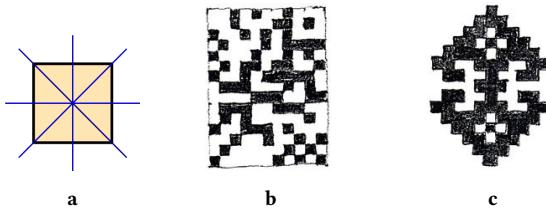
class Shape is common, thus a variation point, for two variants Rectangle and Circle.

## 2.2 Local symmetry and centers

Symmetry is recognized as one of the ideas by which people through the ages have tried to comprehend and create order, beauty, and the perfection of forms [25]. In physics, and generally in natural sciences, the symmetry of an object is defined as a transformation (e.g., reflection, rotation, translation) that leaves the object seemingly unchanged [55], or it is *the immunity to a possible change* [48, 49]. For example, let us consider a square of definite size and orientation as in Figure 1a. The square will remain the same according to eight symmetries, if it is rotated in the plan, about the center, for 0°, 90°, 180°, and 270°, or reflected by a mirror on the shown four axes.

Whenever an overall symmetry is broken, it just creates other local symmetries in the sense that the symmetry is reduced or redistributed, which is different from a total loss of symmetry [48]. For example, ideally, a bilaterally symmetric aircraft should fly straight ahead, but it actually flights in a zigzag way because the flow of air past the aircraft is not bilaterally symmetric. In such way, it must break the symmetry to maintain its stability [55].

<sup>2</sup>The links to the *symfinder* experimental results (screenshots, explanations, online demo) and the *symfinder* public source code are given in Appendix A



**Figure 1: a: The eight symmetries of the square. Two versions of an  $11 \times 15$  array of 69 black and 96 white square blocks - b: a random one and c: a Seljuk pattern [1, p68].**

According to Alexander's theory of centers [2], the order, coherence, and beauty of any structure in nature and human made artifacts is strongly related to local symmetries. Their *geometrical coherence* makes us feel the presence of order, and it can be described in terms of *centers* as building blocks. In this theory, a *center* is not a point, not a perceived center of gravity, it is defined as a *field* of organized force in an object or part of an object which makes that object or part exhibit centrality. For example, in Figure 1b is shown a random arrangement of 69 black and 96 white squares. Because of its incoherence, it is hard even to describe it. Whereas, in Figure 1c these squares have an organized arrangement, known as the Seljuk pattern, which appeared in an old carpet, considered beautiful [1]. Its form of coherence makes it one of the centers in the wholeness of that carpet, which is easy to remember and describe [2, 51].

In the theory of centers, there are around fifteen recurring structural properties that make centers more coherent structures [2, Ch.5]. From those properties, such as levels of scale, boundaries, or alternating repetition, a center is commonly formed by a *local symmetry* [1, pg. 42]. Specifically, the Seljuk pattern in Figure 1c is a center and its coherence is formed by other local centers recursively. Moreover, white squares, which may appear as the background with black ones, have their own (local) symmetries. Centers have been experienced in spatial structures, in nature, buildings, works of art, physics, or psychology [1–3].

Following some other works relating centers, symmetries, and software, discussed in the next section, our contribution makes the main assumption that variation points (*vp*-s) with variants are a kind of *centers of attention and activity* in software design. We thus base our approach on the property of local symmetry for identifying and visualizing potential variability at the implementation level.

### 3 IDENTIFYING VARIABILITY

As stated in the introduction, there are many approaches for detecting variability concepts, especially those for identifying features in code [8, 16, 50], but there is no automated means for identifying *vp*-s with variants in our context of object-oriented techniques [39].

The diversity of these techniques is analysed in different frameworks, taxonomies, and catalogs, by comparing them on different criteria [6, 20, 22, 46, 56]. For instance, in a recent catalog, 16 traditional techniques are compared and classified based on 24 properties [58]. But, despite these comparative schemas, we are not aware that any common property of these techniques exists, and could be used to identify the different kinds of *vp*-s in a uniform way. For example, in Listing 1, the *vp* Shape has a class level granularity and is resolved at runtime, whereas the *vp* draw has a method

level granularity and is resolved at compile time, during product derivation [58]. Both of them resemble two different kinds of *vp*, but with four different properties.

Towards a unified approach for identifying *vp*-s, the majority of traditional language constructs have been shown to be describable in terms of symmetry [12, 66, 67]. In the following we study the property of symmetry in object-oriented techniques, show how it can be interpreted as a local symmetry in reusable code assets, and how this single property can be used to identify all different kinds of *vp*-s.

#### 3.1 Symmetry in software constructs

Symmetry and symmetry breaking have been explored in software, with symmetry in the format of programs, software development life cycles, or search algorithms. Besides, inspired by Alexander's theory of centers [2], symmetry has been identified in different programming language constructs, as well as in software design patterns [12, 24, 65–67]. In the following, we revisit the symmetry in classes, class subtyping, and several design patterns<sup>3</sup>.

*Symmetry in classes.* In object-oriented programming, a class is an extensible code template for creating objects, providing some structure and behavior [64]. At its execution or object instantiation, the definition of its structure and behavior remains unchanged, whereas it enables changes over its instantiated objects. This denotes the symmetry of a class, which can be illustrated on the class Circle in Listing 1 by:

- the *possibility of changes* among all potential circle objects  $c_1, c_2, \dots, c_n$  with different areas and perimeters, and
- the defined computations of area and perimeter in the class Circle that *remain unchanged* for all these objects.

In addition, these objects can be mapped from one another, such as from  $c_1$  to  $c_2$  to  $c_3$ , which represents a substitution as a symmetry transformation. Therefore, a class defines a *substitution symmetry* for its objects.

*Symmetry in subtyping.* In class subtyping, when inheritance is viewed as classification of classes [44, pg.822], all classes of a type path may change, but they must preserve and conform to a common behavior. For example, in Listing 1:

- the *possibility of a change* in the abstract class Shape materializes in its potential different subtypes, such as Rectangle and Circle. Their shown change regards the way the area and perimeter are computed. Whereas,
- the *immunity to change* maps to these subtypes preserving the behavior of their supertype Shape.

Thus, a class subtyping also defines a *substitution symmetry* for its subtypes, which can be substituted as they have the same supertype.

Class subtyping is only one of the ten well-known forms of inheritance [44, pg.822]. According to a coming study, the other forms of inheritance also exhibit the property of symmetry and can be described similarly [13].

*Symmetry in overloading and overriding.* Symmetry appears also in software constructs at method or function level. For instance, method or function overloading lets you define multiple functions

<sup>3</sup>The proof of their symmetry by using the group theory is available elsewhere [66, 67].

of the same name, but with different implementations. For example, for each overloaded method `draw()` of `Rectangle` in Listing 1:

- the number of the taken parameters have *changed* (cf. lines 17–20 and 21–24), whereas
- the name and the return type have remained the same, *unchanged*.

This denotes the symmetry in overloading, where the name of an overloaded function remains unchanged while its arity or types of its parameters change. Thus, overloading also defines a *substitution symmetry* for the overloaded methods, which can be substituted from one to another.

Further, as another construct, method overriding is used to change the behavior of classes under inheritance and it also has a form of symmetry. Specifically, the method overriding name, parameters, and return type, as its signature, remain unchanged while its implementation changes in the subclass by overriding the implementation in the superclass. Thus, method overloading makes somehow possible symmetry in subtyping.

*Symmetry in software design patterns.* For illustration, we now describe the symmetry in three common software design patterns, strategy, factory, and decorator.

Like in most design patterns, strategy uses inheritance [19]. In the strategy pattern, the decision about which algorithm to use is deferred until runtime. It defines a *substitution symmetry*, where the interface for selecting an algorithm remains *unchanged*, whereas the algorithms that enable different behaviors at runtime can be substituted, meaning that they can *change*.

The factory pattern defines an interface with a factory method for creating objects, but lets subclasses decide which class to instantiate. Specifically, concrete creators implement the factory method and create products. We can define it as a specific form of symmetry, namely *factory symmetry*, where the abstract creator and abstract product remain *unchanged*, whereas the concrete creators and products vary.

In the decorator pattern, a set of concrete decorators wrap concrete components, as a means to change their behavior, while their interfaces are preserved. This resembles a *composition symmetry*, where the abstract component and abstract decorator remain *unchanged*, whereas the behavior of concrete components varies, thus *change*, with the concrete decorators.

Such property of symmetry is also evident for most of the other software design patterns, such as the template or observer patterns. Thus, for most of the other language constructs, it has been shown that under a certain transformation, such as substitution in class, behavior, or template symmetry, a specific property of the system is preserved, such as structure, behavior, regularity, similarity, familiarity, or uniformity [65]. This indicates that any of them can be described in terms of symmetry.

In Table 1 we give nine common language features and their elements of symmetries, which are important for automating their identification. They are based on existing studies and the way to interpret symmetry on language features [12, 65–67]. This could be easily extended to include symmetry in other language constructs and design patterns.

**Table 1: Nine language features and their symmetries**

Language feature	Commonality /Unchange	Variability /Change
Class as type	Class/Constructor	Objects
Class subtyping	Superclass/Type	Subclasses
Method overriding	Signature	Classes under
Method overloading	Types of results Structure	Inheritance Signatures
Strategy Pattern	Strategy interface	Algorithms
Factory Pattern	Abstract Creator and product	Concrete creators and products
Decorator Pattern	Components and decorator interfaces	Concrete components and decorators
Template Pattern	Template of a method	Method steps
Observer Pattern	Subject and observer interfaces	Concrete subjects and observers

### 3.2 Identifying variation points with variants

In code reuse, using only classes brings too much symmetry in code, which is perceived as a way to lead to inflexible and rigid programs [65]. Therefore, specifically in object-orientation, the symmetry of programs organized in classes is usually broken by introducing interfaces, abstract classes, and the rise of software design patterns can be seen as a reaction to this problem [12, 65]. Wherever these other mechanisms or techniques are applied, some local symmetries will emerge in code. Therefore, we can infer that the usage of any of them for implementing variability, such as class subtyping, overloading, or design patterns, denotes the existence of a local symmetry in the wholeness of reusable code assets.

We thus build an approach to identify variability by (i) using the fact that each implementation technique is commonly abstracted in terms of a variation point (*vp*) with its variants [58], and then, based on Sections 2.1 and 3.1, (ii) we deduce that a *vp* with variants can be interpreted by the property of local symmetry. Specifically, while *vp*-s resemble the unchanged parts in the design of code assets, variants resemble their changed parts. Hence, as *vp*-s with variants become much more than places where some variability happens, we propose a new definition.

**DEFINITION 1.** *Variation points with variants represent the unchanged and changed parts in software design, are realized by an implementation technique, and abstract the structure (a.k.a., design) and the functionality of the implemented variability. Moreover, they mark the local symmetries in reusable code assets, which resemble centers in Alexander's meaning.*

Based on this definition, to identify variability in terms of *vp*-s with variants, we have to determine the local symmetries in the structure of reusable code assets. For example, the variability in Listing 1 has two local symmetries that can be abstracted as: *vp*\_Shape (lines 1–5) with variants *v*\_Rectangle (lines 6–25) and *v*\_Circle (lines 26–36), and *vp*\_Draw (lines 16–24) with variants *v*\_drawCoordinates (lines 17–20) and *v*\_drawPoint (21–24). The first *vp* resembles the symmetry in inheritance, while the second one the symmetry in overloading. This shows how the different kinds of *vp*-s can be identified by simply identifying the local symmetries in reusable code assets.

In addition to *vp*-s, identifying the variants of a *vp* is important, as they may have nested variability. For example, the class *Rectangle* is a variant of *vp\_Shape* but has a nested *vp* *draw*, which has two other variants. Moreover, all nine language features in Table 1 has a symmetry at the class or method level, indicating that any identified *vp* or variant in these techniques will have a class or method level granularity.

### 3.3 Density of variation points

According to Alexander's theory, the number of local symmetries is crucial for measuring the coherence of a structure [4]. For example, by counting the number of local symmetries it is shown that the Seljuk pattern in Figure 1c has far more local symmetries than the random pattern in Figure 1b [1]. This is then identified as the reason that makes the Seljuk pattern much more coherent, thus easy to recognize, describe, and remember.

Similarly, we propose to use the density of *vp*-s within a code unit (class or file), as a means for locating and describing the most intense places with variability in reusable code assets. First, this is feasible because of the nested nature of *vp*-s, which corresponds to the recursive nature of centers in Alexander's meaning. For example, in Listing 1, the *vp\_Draw* is a nested *vp* of the *vp\_Shape*, by being within one of its variants. This indicates that a larger amount of variability is concentrated in this place. Therefore, we define the density of implemented variability within a code unit as the sum of its *vp*-s, their nested *vp*-s, and the *vp*-s external to the unit that any of its *vp*-s depends on. This density can be simplified and discerned directly from a visualization form of *vp*-s. For example, the density at the class level in Listing 1 has one *vp\_Shape* and one variant *Rectangle*, which shows in minimum one nested *vp*. In this case, the density of Listing 1 is two.

## 4 AUTOMATIC IDENTIFICATION AND VISUALIZATION OF SYMMETRIES

To show the feasibility of our variability identification approach, we developed the *symfinder* toolchain. It enables the automatic identification and visualization of different local symmetries, as described in the previous sections, so that one is helped in determining *vp*-s with variants in a variability-rich system, in visualizing them, and in discerning any pattern of variability by analysis of the density of *vp*-s and variants among different systems.

Figure 2 depicts the whole dockerized toolchain, which consists of three parts, (i) sources fetching from several software projects that are going to be studied, (ii) symmetry identification in the code within the *symfinder* engine, and (iii) visualization through a web browser. The toolchain uses scripts, an engine implemented in Java, and a graph database (Neo4j<sup>4</sup>). It is also deployed within a Docker<sup>5</sup> container so to increase its portability and facilitate its usage.

The source fetching part of the toolkit mainly aims at automating experiments. From a configuration file, the toolchain runs *bash* and *python* scripts in order to fetch sources and checkout the desired tags or commits from some git repositories (*cf.* Figure 2). This enables *symfinder* to work easily over any software system that is publicly available (*e.g.*, on GitHub). Moreover, the main internal

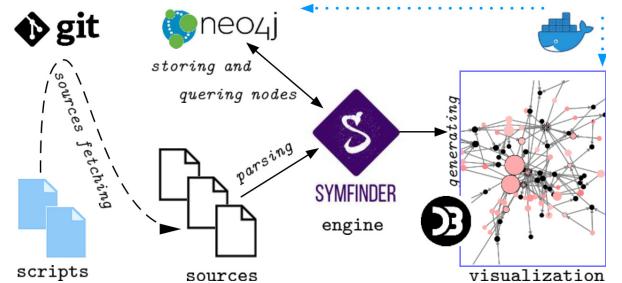


Figure 2: The dockerized *symfinder* toolchain

project structure of *symfinder*, with some usage guidelines, is given in Appendix A.

### 4.1 Identification

At the center of the toolchain is the *symfinder* engine (*cf.* Figure 2), the main purpose of which is to automatically analyse the source code and to build a representation of all potential *vp*-s, (*i.e.*, classes). This process is realized in two main steps. First, the classes of the targeted system are parsed. Targeting in its first version Java-based systems, we reused the Eclipse JDT parser to analyse Java classes. Then, the local symmetries are identified and stored into the Neo4j graph database.

Local symmetries are identified according to the defined symmetry in each language construct, technique, and design pattern given in Table 1. Specifically, each interface, abstract class, extended class, overloaded constructor, and overloaded method is identified. All together, they actually represent the potential *vp*-s. Then, the classes that implement or extend them, including the concrete overloaded constructors and methods are also identified, which should represent respective variants. For example, after parsing the classes in Listing 1, *symfinder* will identify the local symmetry in inheritance among the *vp\_Shape* and its two variant classes *Rectangle* and *Circle*. For each of them, the engine adds a class node and keeps their relationship within the database.

For implementing the previous operations and the following ones, the engine relies on the graph query language of the Neo4j database to identify symmetries. This language, named Cypher<sup>6</sup>, enables the creation and easy querying over nodes, relationships, and properties with patterns covering complex traversals and paths. For example, queries are used to identify the symmetry in the overloaded constructors and methods within each class, as well as to add their number as a property of the class node. In our example, the symmetry in the overloaded *draw* method will be identified within the class *Rectangle* and a value of one will be added to its class node. In addition, the information for the types of class nodes is also saved, whether it was an interface, abstract class, or concrete class. Finally, the *symfinder* engine also identifies the local symmetry in two common software design patterns, strategy and factory. A strategy is identified by its name and by analyzing the structural relationship of classes. The second pattern is identified by its name and by analyzing the return types of methods, a detected factory is a class that contains a method returning an object whose type is a subtype of the declared method return type.

<sup>4</sup><https://neo4j.com/>

<sup>5</sup><https://www.docker.com/>

<sup>6</sup><https://neo4j.com/developer/cypher/>

Node types	Parameters	Visualization
Concrete class ( <i>vp</i> ), Concrete class (Variant with inner <i>vp</i> -s)	Node with black outline	●
Concrete class ( <i>Vari-</i> ant with inner <i>vp</i> -s)	Node without an outline	●
Abstract class ( <i>vp</i> )	Node with dotted outline	●
Interface ( <i>vp</i> )	Black node	●
Constructors ( <i>vp</i> )	Node with shades of red	●
Overloading ( <i>vp</i> )	Node of different size	●
Strategy pattern ( <i>vp</i> )	Node with symbol S	S
Factory pattern ( <i>vp</i> )	Node with symbol F	F
Inheritance	Edge	→

Table 2: The eight kinds of nodes and their relationships used for the visualization of *vp*-s with variants graph

## 4.2 Visualization

After identifying and storing potential *vp*-s with variants into a graph database, we need to provide some means to get more insight regarding the variability aspect of the analysed system. To do so, the *symfinder* toolchain provides the capability to generate an interactive visualization of the elements in the graph (cf. Figure 2).

Instead of visualizing the identified graph of *vp*-s with variants by plain nodes and edges, we considered that it is important to also visualize information regarding the used language constructs, techniques, or design patterns for implementing variability. As in many software and code artifacts visualizations [33, 34, 60, 62, 63], we rely on the visual principles of preattentive perception [15], using some of the seven parameters that can vary in visualization in order to represent data, namely position, size, shape, value (lightness), color hue, orientation, and texture. The six kinds of nodes that we use in *symfinder* for the visualization of the kinds of potential *vp*-s with variants are shown in Table 2.

The D3.js<sup>7</sup> library is used as the visualization support in the *symfinder* toolchain. Although we considered using the visualization capabilities of Neo4j and other visualization forms used in SPL engineering [38], we decided for D3.js as it allows to visualize not only graphs but also a plethora of chart types. We were able to consider them before devising the current form of visualization, and this could also help for future evolutions of the toolchain. Besides, as D3.js visualizations are written in JavaScript, only a web browser is needed, and for *symfinder*, a configuration JavaScript file is only used in a template for the web page that will display the graph.

As an example, Figure 3 shows a visualization excerpt of the identified symmetries in the JavaGeom library [14], a variability-rich system among the ones we used in our experiments (cf. Section 5). It shows the variation point *vp\_AbstractLine2D* and variant *v\_Ray2D*, which forms a comparable variability to the *vp*-s and variants in Listing 1. Specifically, each *vp* node is represented by a circle. A red node with a black outline visualizes a concrete class that is a *vp* (e.g., *vp\_StraightLine2D*). A red node without an outline is a concrete class that is a variant with variability at the method level (e.g., the *v\_Line2D*). A red node with a dotted outline visualizes an abstract class, whereas a black node an interface (e.g., *vp\_LinearElement2D*). Multiple shades of red nodes are used to

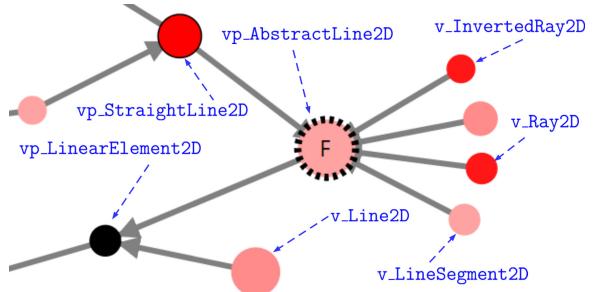


Figure 3: Excerpt of a visualization from the identified symmetries in the JavaGeom library. Annotations in blue are not part of the visualization, they show potential *vp*-s and variant names that are displayed when hovering a node.

visualize the number of constructor overloads for each class or interface. The more overloaded constructors are present, the more intense is the node's color. Next, the size of the node is in function of the number of overloaded methods. For instance, the node *vp\_StraightLine2D* has a more intense red color and bigger size, thus indicating that it has potential variability at both the constructor and method levels. Further, the first letter of a design pattern is used to mark a node that represents that pattern, for example, letter F is used for the factory pattern in *vp\_AbstractLine2D* and its dotted outline denotes its relation to an abstract class. Then, depending on whether nodes are related in design, a directed edge is used to express their relationships, such as in the case of class extension or interface implementation in the current version of the visualization.

With this visualization support, we expect to easily discern, in an analysed system, some zones of interest w.r.t. variability.

## 5 VALIDATION

In order to check whether our tool approach satisfies the identification and visualization of variability, we applied the *symfinder* toolchain on eight object-oriented variability-rich systems. In the following, we present the selected case studies and the obtained results.

### 5.1 Validation case studies

For selecting validation case studies, we considered several criteria, their implementation in Java, the open-source nature of the project, their availability on a git repository, and the fact that they could contain some implemented variabilities.

Geometry related and charting capabilities being typical of some variability to be handled, we first selected JavaGeom, a library for manipulating and processing several families of geometric shapes, already used in other studies [59], JFreeChart, a charting library, and the AWT part of the Java Development Kit. We then selected two projects from the Apache consortium, CXF, a fully featured Web services framework, which could contain variability in its implementation, and Maven, the build automation tool, which architecture is strongly based on plug-ins. We added JUnit 4, as its architecture is based on many design patterns, at least in its previous version 3, and the Java-backend of JHispter, an application

<sup>7</sup><https://d3js.org/>

**Table 3: The eight variability-rich systems and their respective analysed tag or commit ID**

Case study	Url in <a href="https://github.com/">https://github.com/</a>	tag ID	Total LoC	Analysed LoC	# vps	# variants
Java AWT	JetBrains/jdk8u_jdk/src/share/classes/java.awt	jb8u202-b1468	3,514,495	69,974	1,221	1,808
Apache CXF 3.2.7	apache/cxf/core/src/main/java/org/apache/cxf	cxf-3.2.7	810,691	48,655	7,468	9,201
JUnit 4.12	junit-team/junit4/src/main/java	r4.12	30,082	9,317	253	319
Apache Maven 3.6.0	apache/maven	maven-3.6.0	105,342	105,342	1,443	1,393
JHipster 2.0.28	jhipster/jhipster/jhipster-framework/src/main/java	2.0.28	8,035	2,535	140	115
JFreeChart 1.5.0	jfree/jfreechart/src/main/java/org/jfree	v1.5.0	137,074	94,384	1,415	2,103
JavaGeom	dlegland/javaGeom/src	— <sup>a</sup>	33,287	32,755	720	919
ArgoUML	marcusvna/argouml-spl/src	— <sup>b</sup>	178,906	178,906	2,451	3,079

**commit ID:** <sup>a</sup> 7e5ee60ea9febe2acbadb75557d9659d7fafdd28

<sup>b</sup> bcae37308b13b7ee62da0867a77d21a0141a0f18

generator for web applications and microservices, as it has been already used as a variability case study [23]. Similarly, we also selected ArgoUML, a UML diagramming application, used in different studies on SPL engineering [40].

## 5.2 Conducted experiments

We applied the *symfinder* toolchain to analyse and understand the variability of each case study. Details and metrics on the eight case studies are presented in Table 3, with the URL to their public repository, the analysed source package, their analysed tag or commit ID, its total size in lines of code (LoC)<sup>8</sup>, and the size of their analysed source package.

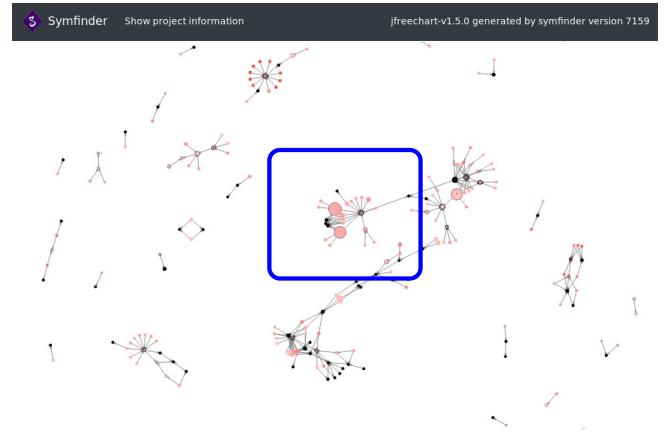
Among the case studies, we experimented with different configurations of *symfinder*. At first, we sought to show that our toolchain can be used to analyse a whole software system or only a desired part of it. For this reason, in some case studies we aimed to identify the variability of the whole software system, such as in Apache Maven 3.6.0, and in some others, of only a single source package, such as the AWT library in the JDK 8 (*cf.* Table 3). Depending on the system, we used commit IDs or tags to grab one specific version, which we have used to tailor the visualization which is presented in this paper. To validate the interoperability of our tool, we made successfully the same experiments in three operating systems, Linux, Mac, and Windows. All the conducted experiments included in this paper are available from <https://deathstar3.github.io/symfinder-demo/>, with extracted screenshots, more explanations on each case, and a deployed online demonstration of the visualization.

## 5.3 Results

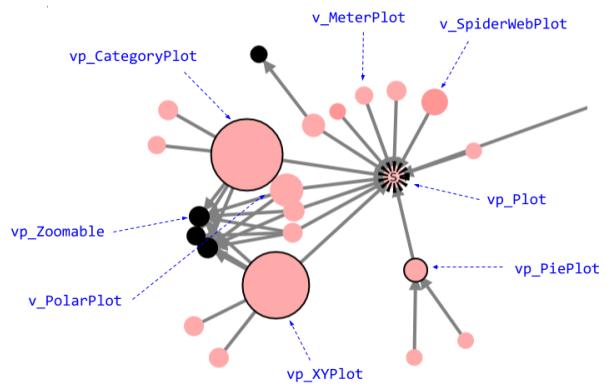
When conducting the experiments, we could successfully visualize the potential variability of each case study and relate it to the used software constructs. For example, in Figure 4 is shown an excerpt from the identified variability in JFreeChart 1.5.0. To ease the reading, the visualization itself can be zoomed in and out, as in Figure 5, and its class name appears when hovering a node. The usage of the visualization also enables us to improve its functionality, as discussed in the following paragraphs.

**5.3.1 Filtering the out of scope vp-s.** Through the analysis of the obtained visualization in each case study, we observed that the larger

<sup>8</sup>For counting the the lines of code we used *gocloc*: <https://github.com/hhatto/gocloc/>

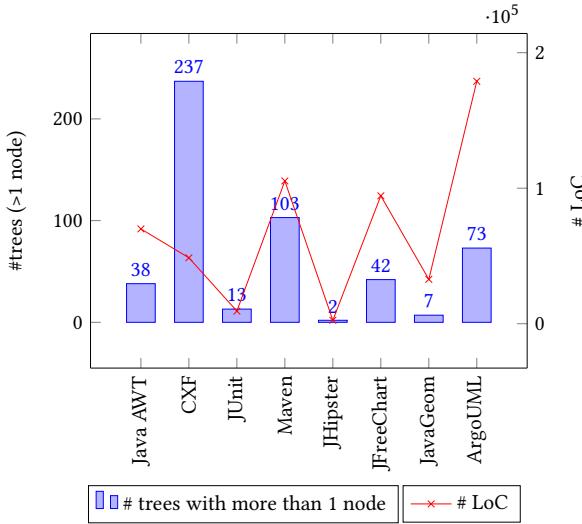


**Figure 4:** An excerpt of the JFreeChart 1.5.0 visualization after removing the out of scope vp-s `org.jfree.chart.event`, `org.jfree.data.general`, and `org.jfree.chart.util.PublicCloneable`



**Figure 5:** The vp-s with variants for the selected zone of interest in Figure 4

number of vp-s and variants may hinder the analysis of a system variability from its visualization. Therefore, we decided to add a



**Figure 6: The number of places with a higher density of *vp*-s and variants at the class level with the # LoC per case study**

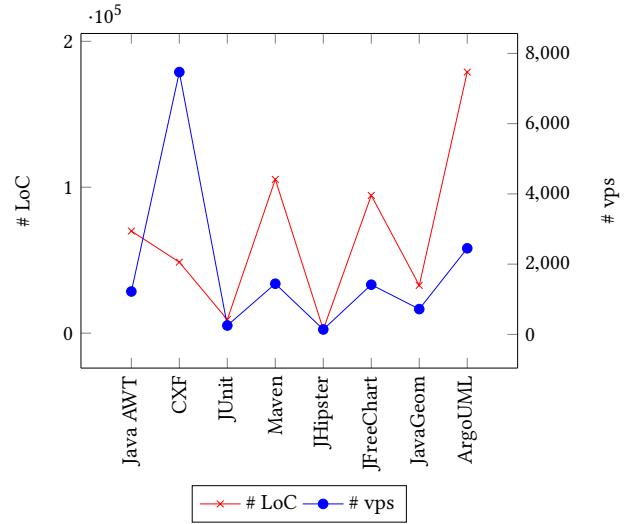
filtering capability in *symfinder*. Currently, filtering is available in the visualization and supports to filter out all the solitary *vp*-s, at once, and also any other individual *vp*, by giving its class name. In Figure 4, filtering is available from the menu "Show project information". This property helped us to analyse and identify several interesting patterns of variability among the targeted systems.

**5.3.2 Understanding the identified variability.** The visualization of variability is mainly a forest-like structure. Therefore, to understand variability, in each case study we focus the analysis on a tree of *vp*-s and variants with method level *vp*-s. In all cases, the visualization helped us in finding as interesting places the trees with a higher density of *vp*-s and variants at the class level (*cf.* Section 3.3). With the appropriate filtering, it was always easy to discern these places.

For example, in JFreeChart, we decided to focus the study in the identified *vp*-s within the blue rectangle in Figure 4. A magnified view of this excerpt of variability is given in Figure 5. Here, the *vp\_Plot* has several variants with method level *vp*-s, such as *v\_PolarPlot*, *v\_MeterPlot*, or *v\_SpiderWebPlot*, which make possible to draw different types of plots in JFreeChart. Then, through a manual trace in code, we observed that the *vp\_Plot* has abstracted the *Plot* class, which is a local symmetry in the strategy pattern.

In the same way, we selected a node in the tree to analyse its method level variability. For example, we observed that the bigger size of the node *vp\_CategoryPlot* corresponds to a large number of symmetries in method overloading in class *CategoryPlot*, which has 29 places with method overloading. Then, the *v\_SpiderWebPlot* has a darker red color as the class *SpiderWebPlot* has a symmetry in constructor overloading with 3 overloaded ones.

We could also easily discern places with the largest or the smallest amount of the factorized commonality and of the constructor level variability. For example, Figure 9a shows an excerpt of the identified variability in the Java 8 AWT library. The node *vp\_Component* has a bigger size than the nodes *vp\_Menu* or *vp\_ItemSelectable*,



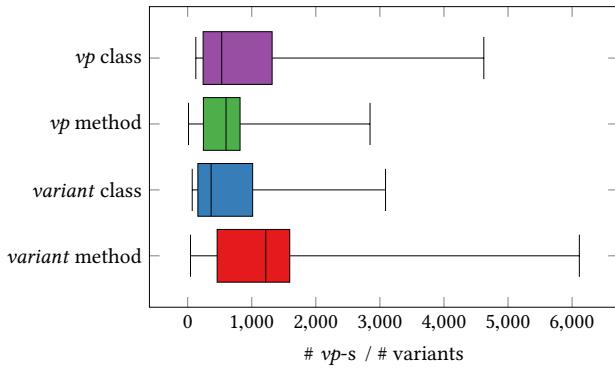
**Figure 7: The correlation of # *vp*-s with the # LoC per case study**

indicating that the *vp\_Component* has a larger amount of commonality for its variants. But, the node *vp\_Window* has a darker red color, indicating that it supports more variability at the constructor level than the *vp\_Component*. Similarly, we analysed each desired *vp* regarding its provided functionality.

To have an overview of variability in each case study, we give in Figure 6 their respective total number of trees. They correspond to the number of places with a density higher than one *vp* or variant at the class level, meaning that the solitary *vp*-s or variants at the class level are excluded from the calculation. For example, JFreeChart contains 42 places with a higher density of *vp*-s and variants with method level variability. Visually, these are the trees with more than a single node in Figure 4. Such a case is the given tree in Figure 5 with 23 nodes. In addition, Figure 6 shows the relationship between the number of trees with higher density and lines of code in each case study.

**5.3.3 The identified number of *vp*-s with variants.** In order to give more insight into the variability of a targeted system, we decided to calculate its identified number of *vp*-s with variants. Interestingly a recent literature review on metrics in SPL engineering shows that the number of *vp*-s is a useful metric for analyzing variability and its implementation in code [18]. It is used to measure the total number of *#ifdef*-blocks when preprocessors are used to implement the variability. Similarly, we used this metric to reason on the size of the implemented variability of our targeted systems. But, in contrast to the existing usage, and in accordance with our *vp* definition (*cf.* Definition 1), the number of *vp*-s now represents the number of local symmetries in reusable code assets, which is complemented with the number of their variants.

The calculation of this metric is automated within the *symfinder* toolchain and is available during the visualization of variability. In Table 3 we give the total number of identified *vp*-s and variants in each case study. Figure 7 shows the correlation between the number of these *vp*-s and the lines of code (LoC) per case study. Further, we give details for the number of *vp*-s and variants at the



**Figure 8: The total number of *vp*-s and variants at the class and method level for the eight case studies**

class and method levels, including the number of *vp*-s and variants at the method and class constructor levels. As the *vp*-s that are related to design patterns overlap with some *vp*-s at class level (*cf.* Section 3.1), we take care to consider them only once during the calculation.

In Figure 8 are summarized the four values of *vp*-s and variants for the eight case studies. We can deduce from them some interesting findings regarding the granularity level of reuse in the observed object-oriented code. There are slightly more *vp*-s at the class level than at the method level, and in the meantime, there are over twice more variants at the method level than at the class level. More globally it seems that both techniques at class and method levels are equally used to implement variability, but we also need to extend the implementation techniques we are able to identify to draw more general conclusions on this.

#### 5.4 Three discerned patterns of variability

From the resulted visualizations, we discerned three patterns of variability that emerge from the different case studies.

As a first pattern, we observed that the bigger size nodes and the darker red nodes appear usually in large trees. For example, the *vp\_XYPlot* in JFreeChart and the *vp\_Component* in Java AWT are two big nodes. Then, the *v\_TimeOut* is a darker red node in JUnit. They are all part of larger trees shown in Figures 5, 9a and 9b, respectively. This indicates that the places with a higher density of variability at the method level have a higher density at the class level, but not conversely. Thus, the cases like in Figure 9b were rare, where *v\_Assert* and *v\_FrameworkAssert* in JUnit 4.12 are two solitary nodes which have a lot of variability at the method level. For this reason, if needed, the single node trees could be filtered out from the visualization.

A second pattern is a way that we can group the eight case studies into (1) those that have a smaller number of trees but a higher density of variability, and (2) those that have a larger number of trees but a lower density of variability. From Figure 6, most of the case studies belong to the first group except the Apache CXF 3.2.7 and Apache Maven 3.6.0 that belong to the second group. These two systems have almost the largest number of trees with more than a single node, but the majority of them are trees with only two or three nodes. For example, Figure 9c shows an excerpt of variability

from Maven. Although this system is highly variable through its plug-in system, nothing is visible in its main project except the interfaces with a single implementation. Specifically, over 90% of its trees have only one *vp* with one or two *vp*-s or variants. For instance, the *vp\_RepositoryRequest* with the *v\_DefaultRepositoryRequest*. Therefore, we used this second pattern to characterize code bases that are more variability-rich, group (1), or less, group (2). As for Maven, it could be interesting to include code from some of its plugins to observe whether some relevant variability zones appear.

The last pattern is revealed in Figure 7. With a single variation in Apache CXF, it shows that the total number of the identified *vp*-s at class and method level seems highly correlated with the number of LoC of a software system. For example, JHipster 2.0.28 and JUnit 4.12 have the smallest number of analysed LoC and the smallest number of *vp*-s. Similarly, ArgoUML is the largest analysed system and has the largest number of *vp*-s.

## 6 DISCUSSIONS

### 6.1 Scope of our study

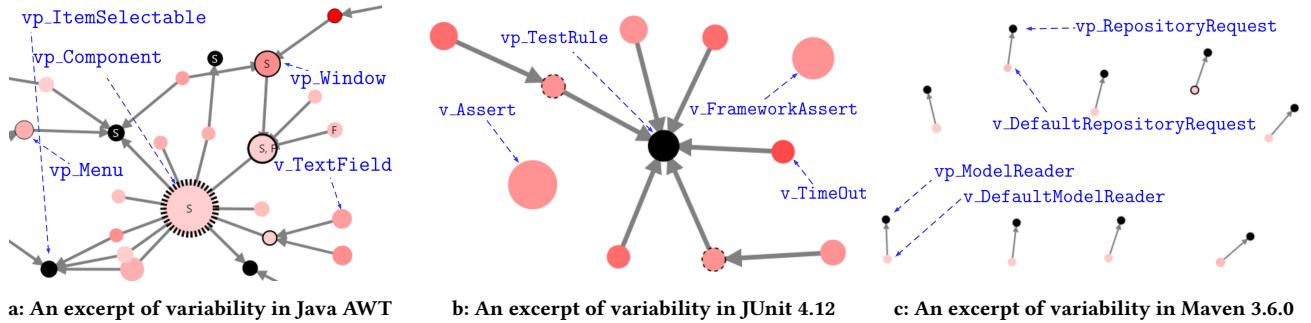
In this study we only considered ten common variability implementation techniques, while variability can be implemented by other language features or paradigms, such as functional programming. Then, some software systems may also vary at the statement level [58], where no technique is really used. However, we decided to consider only the most common variability implementation techniques at the class and method level, which are evident in every object-oriented variability-rich system, and we believe the observed results are sufficient to show the feasibility of the approach. Moreover, we believe that our approach and toolchain can be extended to other used techniques, and at the statement level by using the geometry of code [10, 21], for example with line indentation [45].

### 6.2 Threats to validity

The validity threats we face are related to the *symfinder* toolchain capabilities, and the interpretation of results.

A first threat to validity is on the selected case studies. While the set of case studies is not very large, we have shown it is sufficient to validate the current state of the toolled approach on diverse Java-based software projects. The identification and visualization of symmetries is effective. With a larger set of analysed systems, more or less variability-rich, we believe the obtained results will be similar. Still, we believe that additional systems might highlight some additional variability patterns. This calls for larger experiments as the toolchain itself is extended, as mentioned in the previous paragraph. This is completely in line with our future work plan.

The second threat is on the interpretation of results. First, we explicitly decided to omit the solitary nodes based on the main assertion in the center's theory, where the number of local symmetries resembles the important places in design. But, including them might highlight some additional patterns of variability, some of which can be specific to the domain of the targeted system. For example, 73% and 95% of the trees in JUnit 4.12 and JHipster 2.0.28, respectively, are solitary nodes. We expect that the future enhancements in the approach and toolchain will enable to get more insight on this. Secondly, our experiments show identification of symmetries, and concrete relations between them and some variability



**Figure 9: Example of a: one *vp* with a lot of commonality, b: small trees with high method level variability, and c: large number of nodes with low *vp*-s and variants density**

implementations, but we did not have any complete definition of all present symmetries and variability implementations in the studied code that could have acted as ground truth. Consequently, we only stay at the level of a feasibility demonstration with the current contribution. A first solution would be to examine the whole code of some projects, and it looks feasible for JavaGeom as a starting point. Another one is to find the information in the domain features. Currently, we did not experiment any mapping from existing domain features to identified *vp*-s with variants at the code level. As the ArgoUML case study already has a defined feature model, it could be used for further work on mapping and some measurements on the realizability and usefulness properties [43].

### 6.3 Limitations

Beside the limitations related to the threats discussed above, we see two other limitations in our toolled approach.

First, we currently only analyse Java based systems. Even if it is a widely used object-oriented language, we also observed that some variability can be present in systems or subsystems that are written in JavaScript. This is, for example, the case with JHipster, where we could only analyse the Java backend, which deals only with the generation capabilities. Being able to analyse both languages would enable to study more systems, but also projects architected with different languages, for example, with JavaScript for the frontend, and Java for the backend.

A second limitation is the absence of navigation from *vp*-s or variants into their implementation in code. This can be solved by integrating our toolchain within a development environment, such as Eclipse or IntelliJ, but this is a significant amount of work in implementation and maintenance.

Finally, according to Figure 7, highly variable software systems are likely to have a high number of LoC. Therefore, scalability is an important concern as our toolchain has to be able to analyse large projects. Actually, the analysis of JFreeChart 1.5.0 lasts approximately 25 minutes. In the near future, we will aim at improving the toolchain in order to reduce analysis time, for example, by storing more information in the graph database to reduce the number of analysis passes over the source code.

## 7 RELATED WORK

In reengineering of *clone-and-own* and legacy software systems into SPL, there is a large body of work on feature location and feature identification approaches [8]. Feature location is an activity for recovering the traceability of some pre-existing features to the reusable code assets in an SPL [16, 50]. Whereas, feature identification is an activity for identifying the common and varying units, as potential features, among some related software systems [41, 68]. In both cases, a set of clone-and-own or legacy systems are analysed. In contrast, we consider the class of single variability-rich systems that represent a family of systems but within a single code base. Then, instead of identifying the domain features, for example by doing an intersection of the abstract syntax tree elements of different systems, we identify *vp*-s with variants, as two variability concepts that are closer to the code and abstract the implementation techniques or the reusable design of code assets. Regarding the classification of migration SPL engineering approaches [32], our variability identification process belongs more to the reactive or incremental approaches. Even if we validate it by studying pre-existing systems, we believe that as the *symfinder* toolchain visualizes the identified variability implementations, it can be used to understand and then refactor or incrementally extend the variability of a system under development. Future work with the integration of our toolchain in an IDE would help in exploring this usage.

Approaches for analyzing the variability of preprocessor-based systems seem more closely related to our work [26, 36, 37]. Similarly, we consider a family of systems within a single code base, and study real software. Both approaches are likely to cover a large set of the most used variability implementation techniques in industrial settings. However these works aim at comprehending the usage of C/C++ preprocessor directives for implementing variability, as a single technique, or at extracting them as features into a feature model. On our side, we provide some tool support for understanding the variability of a software system implemented by a set of object-oriented techniques, including design patterns.

Regarding the visualization, a recent mapping study shows that there are several approaches and tools for information visualization in SPL engineering [38]. The most common visualized artifacts are feature models, which use trees or graphs. But, there are very few approaches for visualizing the variability at the code level. The

existing ones use colors [30] or bar diagrams [17]. Some visualizations for feature-file tracing have been also proposed [5], but they are very specific. In general, excluding the configuration process [47, 52], it is well recognized that the majority of the tools in SPL engineering use ad hoc visualization techniques or use the available functionalities inside Eclipse [38]. In contrast, our visualization tends to display, after filtering, trees – which are actually disconnected graphs – conform to the nature of *vp*-s, variants, and their relationships. Displaying classes, inheritance links and some additional metrics, this visualization can be seen as related to the ones for understanding large set of classes, such as polymetric views [33, 34]. However the information we used is just focusing on local symmetries and on the potential implemented variability, but relating other software metrics (e.g. quality metrics) to our set of information is clearly an interesting research topic. Toward that, relations and coupling can be studied with several advanced visualization techniques that are now used for software understanding, such as visualizing large codes as cities [62, 63], as hotspot maps, or as social networks [60].

## 8 CONCLUSION

Many object-oriented variability-rich systems are developed to represent a family of systems but within a single code base. They are also likely to use many different variability implementation techniques (e.g., inheritance, overloading, design patterns), which create in the code assets different kinds of variation points with variants. In this paper, we proposed an identification and visualization method that uses the property of symmetry in software to highlight and abstract different kinds of variation points with variants in a unified way. We relied and extended previous work on software symmetry to systematically map nine object-oriented language features to variability abstractions. Then, we used our prototyped toolchain to identify the corresponding variation points with variants on eight real Java-based systems and provided the first form of visualization to enable software architects to spot zones of interest w.r.t. variability. In addition, we used the density and the number of variation points, as two metrics, to gain more insights for the variability domain of each analysed system. As a result, we discerned three first patterns of variability that characterize the eight variability-rich systems.

We expect this contribution to be a concrete step towards better understanding of variability implementation with traditional techniques, and also to resume the discussion on how to implement the variability within the main decomposition of code. In the future, we first plan to improve the scope of the toolchain regarding the identification of symmetry in other language features, being object-oriented or functional. Then, we plan to integrate with a development environment that will help to automate the navigation from the visualization to code and also map domain features to the identified variability in code. We also plan to analyse and visualize the evolution of the variability implementation patterns in large projects over time and discern new ones. For this reason, we aim at exploiting other software metrics [18]. We also expect to study other properties than symmetry that come from Alexander's theory of centers, aiming to better identify how they could help in understanding large software projects and their variability, such as

using the property of good shape to identify the symmetry at the statement level.

## A APPENDIX

*Current public release.* The latest publicly released source code of the *symfinder* tool, tagged *splc2019-artifact*, is available for download at <https://github.com/DeathStar3/symfinder>.

*symfinder usage guidelines.* In Figure 10 is shown the main project structure of *symfinder*. The numbers on the right side show the sequence of steps to reproduce any of the presented experiments.

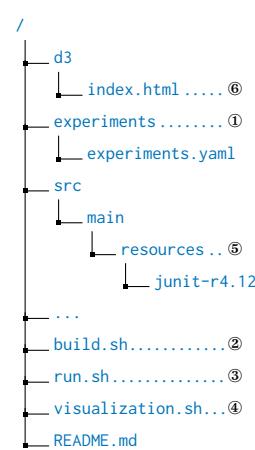


Figure 10: The project structure in *symfinder*

The *README.md* file contains a detailed guide on the technical requirements, how to set up an experiment, to run it, and how to visualize the resulting data for analysis. This guide is valid for three operating systems, GNU/Linux, macOS Sierra 10.12 or newer, and Windows 10 64-bit (Pro, Enterprise or Education). The main requirements for the toolchain are Docker<sup>9</sup> and Docker Compose<sup>10</sup>, so to facilitate the overall portability.

The *experiments.yaml* file in ① is used to set up an experiment. It requires the *git* repository *url* of the targeted system with its tag ID or commit ID, for instance, the *url* of JUnit with tag *r4.12*, given in Table 3. The provided file con-

tains a default configuration that corresponds to all eight analysed systems in Table 3. Still, one can change the configuration to analyse another set of systems.

In ② and ③, *build.sh* and *run.sh* are the main scripts to build and run an experiment. Basically, *run.sh* downloads the sources of the targeted system and starts a Docker Compose environment, whereas *visualization.sh* in ④ generates the visualization data. The downloaded copy of a system is saved locally in the *resources* folder, such as the *junit-r4.12* subfolder in ⑤. Then, *index.html* is used to access the generated visualization of the identified variability for a targeted system (⑥). It can be opened locally using a web browser, through <http://localhost:8181>.

The *README.md* file also contains a visualization example, which is annotated to explain the different elements of visualization.

*symfinder demonstration website.* The generated visualizations of the identified variability for the eight analysed systems are available at <https://deathstar3.github.io/symfinder-demo/>. This site also contains a larger set of examples, enriched with explanations, from the identified variability in each analysed variability-rich system.

<sup>9</sup><https://www.docker.com/>

<sup>10</sup><https://docs.docker.com/compose/>

## REFERENCES

- [1] Christopher Alexander. 1993. *A Foreshadowing of 21st Century Art: The Color and Geometry of Very Early Turkish Carpets* (Center for Environmental Structure, Vol 7). New York: Oxford University Press.
- [2] Christopher Alexander. 2002. *The nature of order: an essay on the art of building and the nature of the universe. Book 1, The phenomenon of life.* Center for Environmental Structure.
- [3] Christopher Alexander. 2002. The process of creating life: Nature of order, Book 2: An essay on the art of building and the nature of the universe. *Berkeley: Center for Environmental Structure* (2002).
- [4] Christopher Alexander and Susan Carey. 1968. Subsymmetries. *Perception & Psychophysics* 4, 2 (1968), 73–77.
- [5] Berima Andam, Andreas Burger, Thorsten Berger, and Michel RV Chaudron. 2017. Florida: Feature location dashboard for extracting and visualizing feature traces. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 100–107.
- [6] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.
- [7] Sven Apel and Dirk Beyer. 2011. Feature cohesion in software product lines: an exploratory study. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 421–430.
- [8] Wesley KC Assunçāo, Roberto E Lopez-Herrejon, Lukas Linsbauer, Silvia R Vergilio, and Alexander Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering* 22, 6 (2017), 2972–3016.
- [9] Rafael Capilla, Jan Bosch, and Kyo-Chul Kang. 2013. *Systems and Software Variability Management*. Springer.
- [10] J Coplien. 1998. Space: the final frontier. *C++ Report* 10, 3 (1998), 11–17.
- [11] James O Coplien. 1999. *Multi-paradigm design for C++*. Vol. 53. Addison-Wesley Reading, MA.
- [12] James O Coplien and Liping Zhao. 2000. Symmetry breaking in software patterns. In *International Symposium on Generative and Component-Based Software Engineering*. Springer, 37–54.
- [13] James O. Coplien and Liping Zhao. 2019. *Toward a general formal foundation of design. Symmetry and broken symmetry*. Technical Report. A VUB Lecture Series Publication. Working draft.
- [14] David Legland. 2019. JavaGeom - Geometry library for Java. <https://github.com/dlegland/javaGeom/tree/master/src> [Online].
- [15] Stephan Diehl. 2007. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media.
- [16] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process* 25, 1 (2013), 53–95.
- [17] Slawomir Duszynski and Martin Becker. 2012. Recovering variability information from the source code of similar software products. In *2012 Third International Workshop on Product Line Approaches in Software Engineering (PLEASE)*. IEEE, 37–40.
- [18] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. 2018. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information and Software Technology* (2018).
- [19] Eric Freeman, Elisabeth Robson, Bert Bates, and Kathy Sierra. 2004. *Head first design patterns*. " O'Reilly Media, Inc".
- [20] Claudia Fritsch, Andreas Lehin, and Thomas Strohm. 2002. Evaluating variability implementation mechanisms. In *Proceedings of International Workshop on Product Line Engineering (PLEES)*. sn, 59–64.
- [21] Richard P Gabriel. 1996. *Patterns of software*. Vol. 62. Oxford University Press New York.
- [22] Critina Gacek and Michalis Anastasopoulos. 2001. Implementing product line variabilities. In *ACM SIGSOFT Software Engineering Notes*, Vol. 26. ACM, 109–117.
- [23] Axel Halin, Alexandre Nuttinck, Mathieu Achet, Xavier Devroey, Gilles Perrouin, and Patrick Heymans. 2017. Yo variability! JHipster: a playground for web-apps analyses. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 44–51.
- [24] Kevin Henney. 2003. The Good, the Bad, and the Koyaanisqatsi. In *Proceedings of the Second Nordic Pattern Languages of Programs Conference, VikingPLoP*, Vol. 2003.
- [25] Weyl Hermann. 1952. *Symmetry*. Princeton University Press.
- [26] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. 2016. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering* 21, 2 (2016), 449–482.
- [27] Ivar Jacobson, Martin Griss, and Patrik Jonsson. 1997. *Software reuse: architecture, process and organization for business success*. ACM Press/Addison-Wesley Publishing Co.
- [28] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.
- [29] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A variability-aware module system. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 773–792.
- [30] Christian Kästner, Salvador Trujillo, and Sven Apel. 2008. Visualizing Software Product Line Variabilities in Source Code.. In *SPLC (2)*, 303–312.
- [31] Maren Krone and Gregor Snelting. 1994. On the inference of configuration structures from source code. In *Proceedings of 16th International Conference on Software Engineering*. IEEE, 49–57.
- [32] CharlesW Krueger. 2001. Easing the transition to software mass customization. In *International Workshop on Software Product-Family Engineering*. Springer, 282–293.
- [33] Michele Lanza and Stéphane Ducasse. 2003. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering* 29, 9 (2003), 782–795.
- [34] Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin Pinzger. 2005. Code-crawler: an information visualization tool for program comprehension. In *Proceedings of the 27th international conference on Software engineering*. ACM, 672–673.
- [35] Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. # ifdef confirmed harmful: Promoting understandable software variation. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 143–150.
- [36] Duc Minh Le, Hyesun Lee, Kyo Chul Kang, and Lee Keun. 2013. Validating consistency between a feature model and its implementation. In *International Conference on Software Reuse*. Springer, 1–16.
- [37] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 105–114.
- [38] Roberto Erick Lopez-Herrejon, Sheny Illescas, and Alexander Egyed. 2018. A systematic mapping study of information visualization for software product line engineering. *Journal of Software: Evolution and Process* 30, 2 (2018), e1912.
- [39] Angela Lozano. 2011. An overview of techniques for detecting software variability concepts in source code. In *International Conference on Conceptual Modeling*. Springer, 141–150.
- [40] Javier Martinez, Nicolas Ordoñez, Xhevahire Térnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. 2018. Feature location benchmark with argoUML SPL. In *Proceedings of the 22nd International Conference on Systems and Software Product Line-Volume 1*. ACM, 257–263.
- [41] Javier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2017. Bottom-up technologies for reuse: automated extractive adoption of software product lines. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 67–70.
- [42] Andreas Metzger and Klaus Pohl. 2014. Software product line engineering and variability management: achievements and challenges. In *Proceedings of the on Future of Software Engineering*. ACM, 70–84.
- [43] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. 2007. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*. IEEE, 243–253.
- [44] Bertrand Meyer. 1988. *Object-oriented software construction*. Vol. 2. Prentice hall New York.
- [45] Richard J Miara, Joyce A Musselman, Juan A Navarro, and Ben Schneiderman. 1983. Program indentation and comprehensibility. *Commun. ACM* 26, 11 (1983), 861–867.
- [46] Thomas Patzke and D. Muthig. 2002. *Product line implementation technologies. Programming language view*. Technical Report 057.02/E. Fraunhofer IESE.
- [47] Andreas Pleuss and Goetz Botterweck. 2012. Visualization of variability and configuration options. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 497–510.
- [48] Joe Rosen. 1995. *Symmetry in science*. Springer.
- [49] Joseph Rosen. 2008. *Symmetry rules: How science and nature are founded on symmetry*. Springer Science & Business Media.
- [50] Julie Rubin and Marsha Chechik. 2013. A survey of feature location techniques. In *Domain Engineering*. Springer, 29–58.
- [51] Nikos Salingeros. 2014. Complexity in architecture and design. *Oz* 36, 1 (2014), 4.
- [52] Denny Schneeweiss and Goetz Botterweck. 2010. Using Flow Maps to Visualize Product Attributes during Feature Configuration.. In *SPLC Workshops*. 219–228.
- [53] Stefan Sobering, Sven Apel, Sergiy Kolesnikov, and Norbert Siegmund. 2016. Quantifying structural attributes of system decompositions in 28 feature-oriented software product lines. *Empirical Software Engineering* 21, 4 (2016), 1670–1705.
- [54] Henry Spencer and Geoff Collyer. 1992. # ifdef considered harmful, or portability experience with C News. (1992).
- [55] Ian Stewart and Martin Golubitsky. 2010. *Fearful symmetry: is God a geometer?* Courier Corporation.
- [56] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. 2005. A taxonomy of variability realization techniques. *Software: Practice and Experience* 35, 8 (2005), 705–754.
- [57] Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton. 1999. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the*

- 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002). IEEE, 107–119.
- [58] Xhevahire Ternava and Philippe Collet. 2017. On the Diversity of Capturing Variability at the Implementation Level. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume B*. ACM, 81–88.
  - [59] Xhevahire Ternava and Philippe Collet. 2017. Tracing Imperfectly Modular Variability in Software Product Line Implementation. In *The 16th International Conference on Software Reuse*.
  - [60] Adam Tornhill. 2015. *Your code as a crime scene: use forensic techniques to arrest defects, bottlenecks, and bad design in your programs*. Pragmatic Bookshelf.
  - [61] C Reid Turner, Alfonso Fuggetta, Luigi Lavazza, and Alexander L Wolf. 1999. A conceptual basis for feature engineering. *Journal of Systems and Software* 49, 1 (1999), 3–15.
  - [62] Richard Wettel and Michele Lanza. 2007. Visualizing software systems as cities. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 92–99.
  - [63] Richard Wettel and Michel Lanza. 2008. Visual exploration of large-scale system evolution. In *2008 15th Working Conference on Reverse Engineering*. IEEE, 219–228.
  - [64] Wikipedia contributors. 2019. Class (computer programming) — Wikipedia, The Free Encyclopedia. [https://en.wikipedia.org/w/index.php?title=Class\\_\(computer\\_programming\)&oldid=884947448](https://en.wikipedia.org/w/index.php?title=Class_(computer_programming)&oldid=884947448) [Online; accessed 26-February-2019].
  - [65] Liping Zhao. 2008. Patterns, symmetry, and symmetry breaking. *Commun. ACM* 51, 3 (2008), 40–46.
  - [66] Liping Zhao and James Coplien. 2003. Understanding symmetry in object-oriented languages. *Journal of Object Technology* 2, 5 (2003), 123–134.
  - [67] Liping Zhao and James O Coplien. 2002. Symmetry in class and type hierarchy. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*. Australian Computer Society, Inc., 181–189.
  - [68] Tewfik Ziadi, Luz Frias, Marcos Aurélio Almeida da Silva, and Mikal Ziane. 2012. Feature identification from the source code of product variants. In *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 417–422.

# Metrics for Analyzing Variability and Its Implementation in Software Product Lines: A Systematic Literature Review

Sascha El-Sharkawy

University of Hildesheim, Institute of  
Computer Science  
Hildesheim, Germany  
elscha@sse.uni-hildesheim.de

Nozomi Yamagishi-Eichler

University of Hildesheim, Institute of  
Computer Science  
Hildesheim, Germany

Klaus Schmid

University of Hildesheim, Institute of  
Computer Science  
Hildesheim, Germany  
schmid@sse.uni-hildesheim.de

## ABSTRACT

This summary refers to the paper *Metrics for analyzing variability and its implementation in software product lines: A systematic literature review*<sup>1</sup>. It was online first in 2018 and was finally published 2019 in the *Information and Software Technology (IST)* journal.

The use of metrics for assessing software products and their qualities is well established in traditional Software Engineering (SE). However, such traditional metrics are typically not applicable to Software Product Line (SPL) engineering as they do not address variability management, a key part of product line engineering. Over time, various specialized product line metrics for SPLs have been described in literature, but no systematic description of these metrics and their characteristics is currently available.

This paper presents a systematic literature review, where we identify metrics explicitly designed for variability models, code artifacts, and metrics taking both kinds of artifacts into account. This captures the core of variability management for product lines. We discovered 42 relevant papers reporting 147 metrics designed for SPLs. We provide a categorization of these metrics and discuss problematic issues regarding their definitions. We also systematically assess the evaluation status of the metrics showing a current lack of high-quality evaluation in the field. Researchers and practitioners can benefit from the published catalog of variability-aware metrics.

## CCS CONCEPTS

- Software and its engineering → Software product lines;
- General and reference → Metrics.

## KEYWORDS

Software Product Lines, SPL, Metrics, Implementation, Systematic Literature Review

## ACM Reference Format:

Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. 2019. Metrics for Analyzing Variability and Its Implementation in Software Product Lines: A Systematic Literature Review. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342368>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7138-4/19/09.

<https://doi.org/10.1145/3336294.3342368>

In our paper *Metrics for analyzing variability and its implementation in software product lines: A systematic literature review*<sup>1</sup>, we present a systematic literature review to identify and characterize variability-aware metrics designed for the needs of SPLs. Our study aims at identifying existing metrics as a basis to draw qualitative conclusions on implementation properties of product lines. We include variability model metrics, because they are linked to all levels of product line realization, including implementation. Thus, we focus only on variability model metrics and code metrics that take variation points into account to characterize product line implementation to answer the following research questions:

**RQ1** Which metrics have been defined for variability models and implementation artifacts of SPLs?

**RQ2** Which correlations between these measures and quality characteristics of product lines have been studied?

We discovered 42 peer-reviewed articles from the last decade to answer **RQ1**. We identified 57 variability model metrics, 34 annotation-based code metrics, 46 code metrics specific to composition-based implementation techniques, and 10 metrics integrating information from variability model and code artifacts. However, only 53 out of 147 metrics ( $\approx 36\%$ ) have been evaluated in 14 out of 42 papers. This indicates that the community has only little knowledge regarding to what extend the existing metrics may be used to draw qualitative conclusions over the studied systems (**RQ2**).

The paper presents a description of the identified metrics, examples, and their intended purpose to provide practitioners and researchers with a catalog of the state of the art of variability-aware implementation metrics. Further, we discuss our key observations, which we made while surveying the literature. These are: There is only a *weak connection to established metrics* from traditional software engineering, the *weak use of related work* leads to redundant definitions of similar metrics, there are *ambiguous metric definitions* and *problematic evaluations* are used. Finally, we were surprised that there exist only a very *small number of metrics combining variability information* from multiple sources. We found only one empirical analysis evaluating such metrics combining the information from variability model and implementation artifacts. This study indicates a high usefulness of such a combined metric, encouraging further research along these lines.

## ACKNOWLEDGMENTS

This work is partially supported by the ITEA3 project REVaMP<sup>2</sup>, funded by the BMBF (German Ministry of Research and Education) under grant 01IS16042H. Any opinions expressed herein are solely by the authors and not of the BMBF.

<sup>1</sup><https://doi.org/10.1016/j.infsof.2018.08.015>

# Semantic Evolution Analysis of Feature Models

Imke Drave

Software Engineering, RWTH Aachen University  
Aachen, Germany

Judith Michael

Software Engineering, RWTH Aachen University  
Aachen, Germany

Oliver Kautz

Software Engineering, RWTH Aachen University  
Aachen, Germany

Bernhard Rumpe

Software Engineering, RWTH Aachen University  
Aachen, Germany

## ABSTRACT

During the development process, feature models change continuously. Analyzing the semantic differences between consecutive feature model versions is important throughout the entire development process to detect unintended changes of the modeled product line. Previous work introduced a semantic differencing technique for feature models based on a closed-world assumption, which reveals the differences between two feature models when allowing products to only contain features used in the models. However, this does not reflect the stepwise refinement of feature models in early development stages. Therefore, we introduce an open-world semantics, an automatic method for semantic differencing of feature models with respect to the novel semantics, and formally relate the open- and closed-world semantics. We formally prove our results, including the relation between the different semantics as well as the soundness and completeness of the semantic differencing procedure. In conjunction with previous work, the results enable effective semantic feature model evolution analyses throughout the entire development process.

## CCS CONCEPTS

- Software and its engineering → Software product lines.

## KEYWORDS

feature modeling, model evolution analysis, semantic differences, open-world semantics

### ACM Reference Format:

Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. 2019. Semantic Evolution Analysis of Feature Models. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3336294.3336300>

## 1 INTRODUCTION

Highly differentiated customer requirements challenge product selection, production, and delivery not only in software engineering,

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336300>

but also in, e.g., production and logistics companies. Thus, configuring product families is nowadays standard in many business sectors, as it improves customer satisfaction and helps to achieve cost reduction. Configurations are not restricted to a certain domain or application and are applicable to any product with a high demand for variability, such as cars, bikes, laptops, and cameras. For these products, customers demand the ability to choose product features in a selection process provided by an online shop or an order process. In industry, software product line (SPL) engineering [14, 40] became a meaningful way to handle the development of a diversity of similar software applications [13, 17, 27, 44]. Feature modeling integrates the feature-oriented paradigm [4] for SPL development with model-driven engineering [20] by means of feature models (FMs) that represent all possible product configurations [22], similar to the 150% modeling approach [22].

In model-driven software engineering (MDSE), models are the primary development artifacts [20] and, thus, evolve over time. Refinement is an essential concept in MDSE. A model is a *refinement* of another model, if the semantics of the latter subsumes the semantics of the former [23]. Refinement steps in MDSE, i.e., changing a model such that the new version is a refinement of the former version, are naturally performed in reaction to changing requirements and availability of additional information [26]. The idea is to start with an underspecified model encoding the available information and to iteratively refine the model once additional information becomes available, until ultimately obtaining a correct system implementation. Effective model evolution management is essential to detect bugs and explore design alternatives. Syntactic model evolution management is fairly established in modern MDSE development (e.g., [3, 28–31, 45, 46]). Detecting semantic differences between two models enables to verify refinement with respect to the model's meaning and is subject to ongoing research (e.g., [2, 12, 18, 19, 32, 33, 36–38]). Recent approaches combine syntactic with semantic differencing to identify the impact of syntactic changes on a model's semantics [21, 26, 34].

FMs describe features and the relations (mandatory, optional, alternative, exclusive, implies, excludes) between them. The relations and further propositional constraints restrict the set of valid configurations [51, 52]. The semantics of a FM is the set of all configurations that respect all the restrictions of the FM. Existing approaches for semantic differencing of FMs [2] provide meaningful results under a *closed-world assumption* [41]. In the context of FMs, the closed-world assumption considers features not used in a FM to not exist. However, this assumption yields a quite restrictive semantics, especially in the context of model evolution analysis. To illustrate this, consider a restaurant offering several fixed dishes

on a menu. In a closed-world, individual customer wishes, such as exchanging a supplement of a dish by a supplement of another one, would not be considered as an allowed configuration of the menu.

The *open-world assumption* considers FM semantics in a less restrictive way: Features that are not used in a FM are not considered to be non-existent, rather, it is assumed that their instantiation is not restricted. In the context of the restaurant example, in the open-world, exchanging one supplement by any other one, be it part of another dish or made up by the customer, would be considered a valid dish. In this case, it is more appropriate to stick to the open-world semantics, as it is less restrictive and thereby reflects the customer's intuition when choosing the dish. The fact that a dish is not explicitly mentioned on the menu does not mean that the restaurant's chef is not able to cook it. The open-world semantics as proposed in this paper should not replace the usual closed-world FM semantics but is to be seen as a complement to it, especially for performing analyses in early development stages.

In the following, Section 2 shows practical examples to motivate our idea and to illustrate the difference between closed- and open-world semantics. Section 3 introduces an abstract syntax of FMs, defines closed- and open-world semantics of FMs and analyses their relation. Section 4 introduces semantic FM differencing. Section 5 discusses our approach and its use in early development stages. Section 6 relates our approach to previous work concerning automated reasoning, semantic property analyses as well as model composition and synthesis. The last section concludes this paper and offers ideas for future research.

## 2 EXAMPLES

This section presents examples that illustrate potential use cases where the open-world semantics is useful and that illustrate the difference between the closed- and open-world semantics.

A software developer was asked to create an online ordering service for a pizzeria. Therein, customers should be able to choose ingredients individually. During requirements engineering, the pizzeria owner and the web developer discuss the pizza configurations to be offered online. The developer creates a FM that represents the available pizza configurations (*cf.* Figure 1, FM  $p_1$ ): A pizza needs at least a base. The base can be either wheat or gluten-free. Optionally, customers can choose a sauce (tomato, hollandaise or nutella) and several toppings (banana, mozzarella, salami, broccoli). The implementation of all possible configurations in the online shop is based on the FM. Nevertheless, customers know the pizzeria quite well: They know, if other toppings are available at the restaurant, like ham, mushrooms, pineapple or rocket salad, it is possible to order them as well, via an additional phone call. In the real world, the FM modeling the pizzas available at the online store has an open-world semantics: Ingredient combinations, which are not explicitly forbidden by the FM, make up a pizza that can be ordered at the online shop.

The owner would like to make ham an “official” topping by offering it online as well, so the developer adds it as an optional feature of topping. Also, lately the pizzeria had been approached by several customers asking for sausage crust pizza, which is also added as an optional feature to the FM. Moreover, the cook refuses to put banana on any other sauce than nutella, which the developer

implements by a requires-relation between banana and nutella. The FM  $p_{1.2}$  in Figure 1 displays the adapted model.

However, the sales figures for sausage crust do not develop as expected. Thus, the owner instructs the developer to change it to cheesy crust instead (*cf.*  $p_{1.3}$  in Figure 1). The tables in Figure 2 depict pizza configurations illustrating the difference between closed- and open-world semantics. The configuration  $c_1 = \{\text{pizza, base, wheat, topping, salami, ham}\}$  is contained in the open-world semantics of  $p_1$  but not in its closed-world semantics. The configuration  $c_1$  is, however, an element of the closed-world semantics of  $p_{1.2}$  and, therefore, a diff witness for the closed-world semantic difference from  $p_{1.2}$  to  $p_1$ . In the open world,  $p_1$  does not restrict including the ham-feature in valid configuration, therefore, ham-pizzas are available. Using a closed-world semantics, no ham-topped pizza are available in the  $p_1$  based web-shop. Since customers were able to order ham pizza while  $p_1$  implemented the available configurations, the closed-world semantics do not represent all available products. The Venn diagram depicted in Figure 3 illustrates that, indeed,  $p_{1.2}$  and  $p_{1.3}$  are both open-world refinements of  $p_1$ , while in the closed-world, all three have pairwise incomparable semantics. The former corresponds to the owner's and the developer's intuition of how the product range changes after adding the ham-topping and sausage crust-feature as well as the “requires” constraint between the nutella-sauce and the banana-topping. However, even in the open-world semantics,  $p_{1.3}$  is no refinement of  $p_{1.2}$  and vice versa. The configuration  $c_2 = \{\text{pizza, base, wheat, sauce, tomato, topping, banana}\}$  (*cf.* Figure 2) is a diff witness for both, the open-world and the closed-world semantic difference from  $p_1$  to  $p_{1.2}$ . It is an element of the open- as well as the closed-world semantics of  $p_1$ , whereas it is no element of both, the open-world and the closed-world semantics of  $p_{1.2}$ . The “requires” constraint between the banana topping and the nutella sauce causes the witness. Using the closed-world semantics, the configuration  $c_3 = \{\text{pizza, base, wheat, cheesy crust}\}$ , is valid in FM  $p_{1.3}$  but not valid in FM  $p_{1.2}$ . Using the open-world semantics, it is a valid configuration of both FMs. However, as Figure 3 shows, the two have incomparable open- and closed-world semantics.

The following sections define an abstract syntax for FMs and formalize both semantics and the difference between them.

## 3 FEATURE MODELS

This section introduces an abstract syntax for FMs and formally defines an open-world FM semantics. Further, it recaps the well-known closed-world FM semantics and relates the two semantics with each other.

### 3.1 An Abstract Syntax for Feature Models

Let  $U_F$  denote a countable (possibly infinite) universe of features. The elements of the set  $U_F$  represent feature names. The abstract syntax of FMs is defined as follows (similar to [51, 52]):

**DEFINITION 1 (FEATURE MODEL).** *A feature model is a tuple  $FM = (F, E, r, child, I, EX)$  where*

- $F \subseteq U_F$  is a finite set of features,
- $(F, E, r)$  is a directed rooted tree with nodes  $F$ , root  $r \in F$ , and edges  $E \subseteq F \times F$ ,
- $r$  is the root feature,

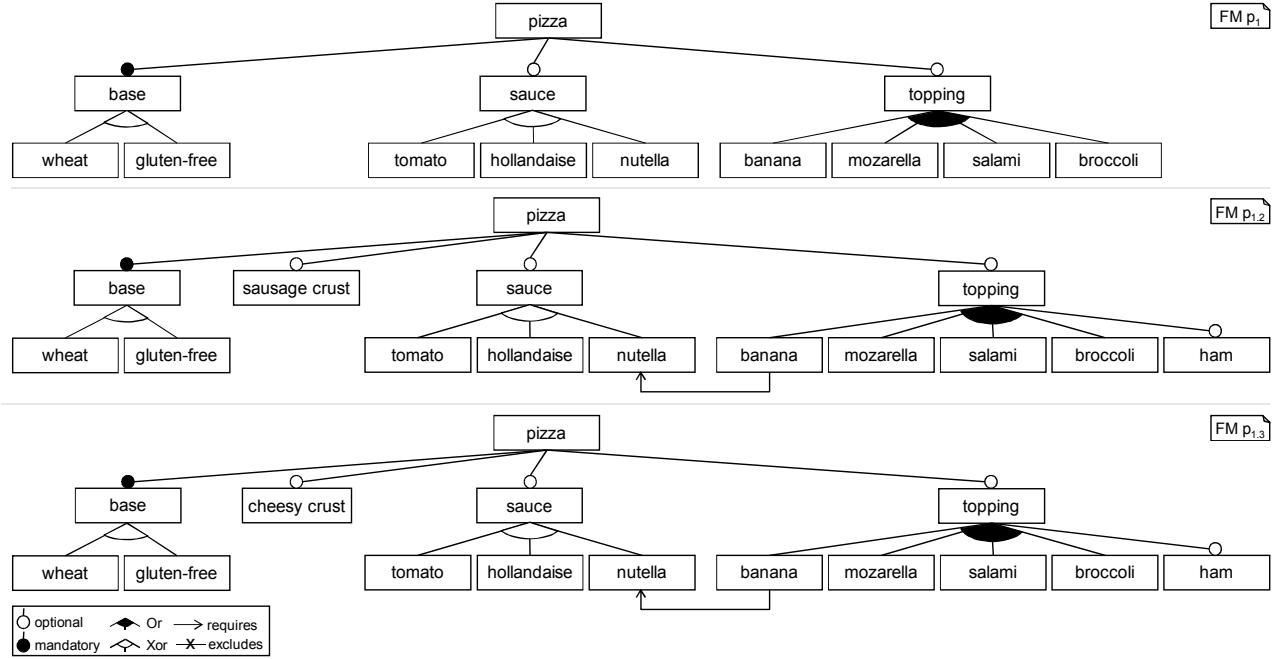


Figure 1: Three consecutive versions of the FM modeling pizzas available at the pizza store.

Closed World Semantics:

Configuration	$\in \llbracket p_1 \rrbracket^{cw}$	$\in \llbracket p_{1.2} \rrbracket^{cw}$	$\in \llbracket p_1 \rrbracket^{cw} \setminus \llbracket p_{1.2} \rrbracket^{cw}$	$\in \llbracket p_{1.2} \rrbracket^{cw} \setminus \llbracket p_1 \rrbracket^{cw}$
$C_1 = \{\text{pizza, base, wheat, topping, salami, ham}\}$	✗	✓	✗	✓
$C_2 = \{\text{pizza, base, wheat, sauce, tomato, topping, banana}\}$	✓	✗	✓	✗

Open World Semantics:

Configuration	$\in \llbracket p_1 \rrbracket^{ow}$	$\in \llbracket p_{1.2} \rrbracket^{ow}$	$\in \llbracket p_1 \rrbracket^{ow} \setminus \llbracket p_{1.2} \rrbracket^{ow}$	$\in \llbracket p_{1.2} \rrbracket^{ow} \setminus \llbracket p_1 \rrbracket^{ow}$
$C_1 = \{\text{pizza, base, wheat, topping, ham}\}$	✓	✓	✗	✗
$C_2 = \{\text{pizza, base, wheat, sauce, tomato, topping, banana}\}$	✓	✗	✓	✗

Figure 2: Configurations in the closed- and open-world semantics of the FMs depicted in Figure 1.

- $\text{child} : F \rightarrow \wp(\wp(F))$  maps each feature  $f \in F$  to its possible sets of children  $\text{child}(f) \subseteq \wp(\{c \in F \mid (f, c) \in E\})$ .
- $I \subseteq F \times F$  is a set of implies constraints,
- $EX \subseteq F \times F$  is a set of excludes constraints.

For each  $f \in F \setminus \{r\}$ , we denote by  $p_{FM}(f) \in F$  the parent of  $f$  in the feature model  $FM$ , i.e., for all features  $f \in F \setminus \{r\}$ , we define  $p_{FM}(f) = p$  iff  $(p, f) \in E$ . If  $FM$  is known from the context, we simply write  $p(f)$  instead of  $p_{FM}(f)$ .

Usually, the feature trees modeled in graphical notation are called feature diagrams and the combination of a feature diagram with a propositional formula is called FM (e.g., [2]). For a feature  $f \in F$ ,

the set  $\text{child}(f)$  is not required to have a graphical feature-group (e.g., Or, Xor) representation. This enables to describe every feature diagram in the common graphical representation and additional constraints, for example, modeled with propositional formulas. Figure 4 depicts an example FM that can be formally defined as  $(F, E, A, \text{child}, I, EX)$  with

- the set of features  $F = \{A, B, C, D, E, F, G\}$ ,
- the root feature  $A$ ,
- the set of edges  $E = \{(A, B), (A, C), (B, D), (B, E), (C, F), (C, G)\}$ ,
- the function  $\text{child} = \{A \mapsto \{\{B\}, \{B, C\}\}, B \mapsto \{\{D\}, \{E\}\}, C \mapsto \{\{F\}, \{G\}\}, D \mapsto \emptyset, E \mapsto \emptyset, F \mapsto \emptyset, G \mapsto \emptyset\}$ ,

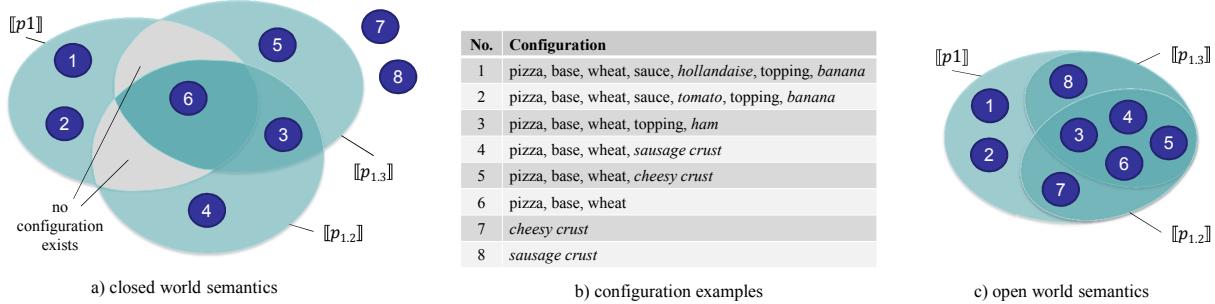


Figure 3: Closed- and open-world semantics for the pizza examples in Fig. 1

- the set of implies constraints  $I = \{(F, E)\}$ , and
- the set of excludes constraints  $EX = \{(D, G)\}$ .

### 3.2 Feature Model Semantics

The semantics of FMs is defined in terms of sets of configurations.

**DEFINITION 2 (FEATURE CONFIGURATION).** A (feature) configuration is a finite set of features  $C \subseteq U_F$ .

The following defines when a configuration is considered to be possible in a FM with respect to the closed- and open-world semantics. We distinguish between configurations that are *products* of a FM and configurations that are *valid* in a FM. Intuitively, a configuration is valid in a FM, if it does not violate any constraint induced by the FM. A configuration is a product of a FM, if it does not violate any constraints induced by the FM and contains only features that are also contained in the FM. Consequently, every product of a FM is also valid in the FM. The valid configurations of a FM are the elements of its open-world semantics:

**DEFINITION 3 (OPEN-WORLD SEMANTICS).** A configuration  $C \subseteq U_F$  is said to be valid in a FM  $FM = (F, E, r, child, I, EX)$  iff

- (1)  $\forall f \in C \cap F : f \neq r \Rightarrow p(f) \in C$ ,
- (2)  $\forall f \in C \cap F : \{g \in C \mid p(g) = f\} \in child(f)$ ,
- (3)  $\forall (f, g) \in I : f \in C \Rightarrow g \in C$ ,
- (4)  $\forall (f, g) \in EX : f \in C \Rightarrow g \notin C$ .

The open-world semantics  $\llbracket FM \rrbracket^{ow}$  of FM is defined as the set of all configurations that are valid in FM.

The first condition requires that if the configuration contains a non-root feature of the FM, then the configuration must also contain the feature's parent feature. Valid configurations are not required to contain the root feature. The second condition states that all maximal sets of features with a common parent are possible sets of child features of the parent features. The last two conditions state that requires and excludes constraints must be respected.

The closed-world semantics for FMs (e.g., [2, 5, 8, 16, 55]) requires that all features of all configurations in the semantics of a FM must be contained in the feature set of the FM. Thus, the closed-world semantics are more constraining than the open-world semantics. The *products* of a FM are the configurations that are derived from the features contained in the FM and the constraints imposed by the FM. The closed-world semantics is formally defined as follows:

**DEFINITION 4 (CLOSED-WORLD SEMANTICS).** A configuration  $C \subseteq U_F$  is called a product of a FM  $FM = (F, E, r, child, I, EX)$  iff

- (1)  $C \subseteq F$  and
- (2)  $C$  is valid in FM.

The closed-world semantics  $\llbracket FM \rrbracket^{cw}$  of FM is defined as the set of all products of FM.

The first condition in Definition 4 requires that all features of a FM's product must be elements of the FM. Thus, if a feature does not exist in a FM, then the feature is also assumed to be non-existent in any configuration of the FM's closed-world semantics. In contrast, using the open-world semantics, the feature is considered to be unconstrained. The difference between the closed-world semantics and the open-world semantics seems to be small. However, the choice of the semantics has strong implications in the context of model evolution analysis as discussed in Section 4.

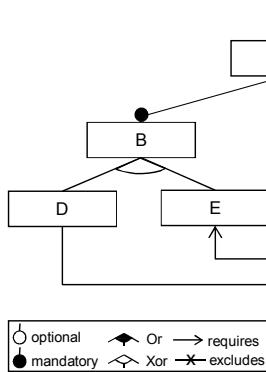
Each product of the FM depicted in Figure 4 may only contain features of the set  $F = \{A, B, C, D, E, F, G\}$ . It must not contain other features. Configurations containing the feature H, for instance, are no valid products of the FM according to Definition 4. For example, the configuration  $\{A, B, D\}$  is a product of the FM. It satisfies all constraints induced by the FM and all features in the configuration are also used in the FM. In contrast, the configuration  $\{A, B, E, H\}$  is valid in the FM but no product of the FM: Although the configuration satisfies all constraints induced by the FM, it does not exclusively contain features that are used in the FM. The configuration  $\{A, B, C, D, G\}$  is neither valid in the FM nor a product of the FM, because it violates the FM's excludes constraint.

Considering the open-world semantics, features not used in a FM are unconstrained by the FM. Thus, removing features that are not used in an FM from a valid configuration of the FM yields a valid configuration of the FM.

**LEMMA 1.** Let  $FM = (F, E, r, child, I, EX)$  be a feature model. Further, let  $C \subseteq U_F$  and  $C' \subseteq C$  be two configurations with  $C \cap F = C' \cap F$ . If  $C \in \llbracket FM \rrbracket^{ow}$ , then  $C' \in \llbracket FM \rrbracket^{ow}$ .

**PROOF.** Let  $C, C'$  and  $FM$  be given as above. Assume  $C \in \llbracket FM \rrbracket^{ow}$ . Then, by Definition 3, the following is satisfied:

- (1)  $\forall f \in C \cap F : f \neq r \Rightarrow p(f) \in C$ ,
- (2)  $\forall f \in C \cap F : \{g \in C \mid p(g) = f\} \in child(f)$ ,
- (3)  $\forall (f, g) \in I : f \in C \Rightarrow g \in C$ ,
- (4)  $\forall (f, g) \in EX : f \in C \Rightarrow g \notin C$ .



$$F = \{A, B, C, D, E, F, G\} \quad r = A \quad I = \{(F, E)\} \quad EX = \{(G, D)\}$$

Configuration	$\in \llbracket F \rrbracket^{cw}$	$\in \llbracket F \rrbracket^{ow}$	$\in \llbracket F \rrbracket^{cw} \cap \llbracket F \rrbracket^{ow}$
$C_1 = \{A, B, D\}$	✓	✓	✓
$C_2 = \{A, B, D, C, G\}$	✗	✗	✗
$C_3 = \{A, B, D, C, H\}$	✗	✓	✗

Figure 4: Feature model containing all quintessential modeling elements.

(a) As  $C \cap F = C' \cap F$  and  $p(f) \in F$  for all  $f \in F$ , (1) implies  $\forall f \in C' \cap F : f \neq r \Rightarrow p(f) \in C$ .

(b) As  $C \cap F = C' \cap F$  and  $p(f) \in F$  for all  $f \in F$ , we have  $\{g \in C \mid p(g) = f\} = \{g \in C' \mid p(g) = f\}$ . Using this and (2), we obtain that  $\forall f \in C' \cap F : \{g \in C' \mid p(g) = f\} \in \text{child}(f)$ .

(c) As  $C \cap F = C' \cap F$  and  $I \subseteq F \times F$ , (3) implies  $\forall (f, g) \in I : f \in C' \Rightarrow g \in C'$ .

(d) As  $C \cap F = C' \cap F$  and  $EX \subseteq F \times F$ , (4) implies  $\forall (f, g) \in EX : f \in C' \Rightarrow g \notin C'$ .

From (a)-(d) and Definition 3, we can conclude  $C' \in \llbracket F \rrbracket^{ow}$ .

□

Similarly, adding features that are not contained in a FM to a valid configuration of the FM yields a valid configuration.

LEMMA 2. Let  $FM = (F, E, r, \text{child}, I, EX)$  be a feature model. Further, let  $C \subseteq U_F$  and  $N \subseteq U_F \setminus F$  be two configurations. If  $C \in \llbracket F \rrbracket^{ow}$ , then  $C \cup N \in \llbracket F \rrbracket^{ow}$ .

PROOF. Let  $FM, C$ , and  $N$  be given as above.

Assume  $C \in \llbracket F \rrbracket^{ow}$ . As  $N \cap F = \emptyset$ , we have that  $(C \cup N) \cap F = C \cap F$ . As  $\forall g \in F : p(g) \in F$  and since  $N \cap F = \emptyset$ , it holds that  $\{g \in C \mid p(g) = f\} = \{g \in C \cup N \mid p(g) = f\}$  for all  $f \in F$ . With  $C \in \llbracket F \rrbracket^{ow}$  and using the above, we obtain

(a)  $\forall f \in (C \cup N) \cap F : f \neq r \Rightarrow p(f) \in (C \cup N)$  and

(b)  $\forall f \in (C \cup N) \cap F : \{g \in (C \cup N) \mid p(g) = f\} \in \text{child}(f)$ .

As  $N \cap F = \emptyset$ , we have that  $f \in C \cup N \Leftrightarrow f \in C$  for all  $f \in F$ . As further  $C \in \llbracket F \rrbracket^{ow}$ , we obtain

(c)  $\forall (f, g) \in I : f \in (C \cup N) \Rightarrow g \in (C \cup N)$  and

(d)  $\forall (f, g) \in EX : f \in (C \cup N) \Rightarrow g \notin (C \cup N)$ .

From (a)-(d) we can conclude that  $C \cup N \in \llbracket F \rrbracket^{ow}$ .

□

### 3.3 On the Relation between the Closed World Semantics and the Open World Semantics

The open-world semantics is more liberal than the closed-world semantics: Every product of a FM is also a valid configuration of the FM. The other direction does not hold.

LEMMA 3.  $\llbracket F \rrbracket^{cw} \subseteq \llbracket F \rrbracket^{ow}$  for all feature models  $FM$ .

PROOF. Let  $FM$  be a FM and let  $C \in \llbracket F \rrbracket^{cw}$ . By definition,  $C$  is valid in  $FM$ . Thus,  $C \in \llbracket F \rrbracket^{ow}$ .

□

Independent of the feature universe  $U_F$ , the closed-world semantics of every FM is always finite. If the universe of features  $U_F$  is infinite, then the open-world semantics of a FM is always infinite. Thus, enumerating all the products of a FM is always possible, whereas enumerating all of the FM's valid configurations is usually not possible. In contrast, if the universe of features  $U_F$  is finite, then the open-world semantics of every FM is also finite. In the special case where the feature universe  $U_F$  is finite and the FM uses all features of the universe, the open-world semantics and the closed-world semantics of the FM coincide.

LEMMA 4. For every feature model  $FM = (F, E, r, \text{child}, I, EX)$ , the following statements hold:

- (1)  $\llbracket F \rrbracket^{cw}$  is a finite set.
- (2) If  $U_F$  is infinite, then  $\llbracket F \rrbracket^{ow}$  is infinite.
- (3) If  $U_F$  is finite, then  $\llbracket F \rrbracket^{ow}$  is finite.
- (4) If  $U_F = F$ , then  $\llbracket F \rrbracket^{cw} = \llbracket F \rrbracket^{ow}$ .

PROOF. Let  $FM = (F, E, r, \text{child}, I, EX)$  be a FM.

(1) By Definition 4, for every configuration  $C \in \llbracket F \rrbracket^{cw}$ , it holds that  $C \subseteq F$ . Thus,  $\llbracket F \rrbracket^{cw} \subseteq \varphi(F)$ . As  $F$  is finite,  $\varphi(F)$  is finite, and thus  $\llbracket F \rrbracket^{cw}$  is finite.

(2) Assume  $U_F$  is infinite. The set of features  $F$  is by definition finite. Thus, the set  $U_F \setminus F$  of features that are not used in the feature model  $FM$  is infinite. Using Definition 3, it is easy to verify that  $\forall f \in U_F \setminus F : \{f\} \in \llbracket F \rrbracket^{ow}$ , i.e., each configuration containing one feature not used in  $FM$  is valid in  $FM$ . As  $U_F \setminus F$  is infinite, this implies  $\llbracket F \rrbracket^{ow}$  is infinite.

(3) Assume  $U_F$  is finite. Then,  $\varphi(U_F)$  is finite. As  $\llbracket F \rrbracket^{ow} \subseteq \varphi(U_F)$ , we can conclude that  $\llbracket F \rrbracket^{ow}$  is finite.

(4) Assume  $U_F = F$ . In particular, this implies that  $U_F$  is finite since  $F$  is finite by definition. By Lemma 3 we have that  $\llbracket F \rrbracket^{cw} \subseteq \llbracket F \rrbracket^{ow}$ . It remains to show that  $\llbracket F \rrbracket^{ow} \subseteq \llbracket F \rrbracket^{cw}$ . Let  $C \in \llbracket F \rrbracket^{ow}$  be a valid configuration of  $FM$ . As  $U_F = F$  and  $C \subseteq U_F$ , we have that  $C \subseteq F$ . As  $C$  is valid in  $FM$  and  $C \subseteq F$ , we can conclude with Definition 4 that  $C \in \llbracket F \rrbracket^{cw}$ .

Further, the open-world semantics is antitone (order-reversing) with respect to the elements modeled in the FMs: The addition of information to a FM further restricts its set of valid configurations.

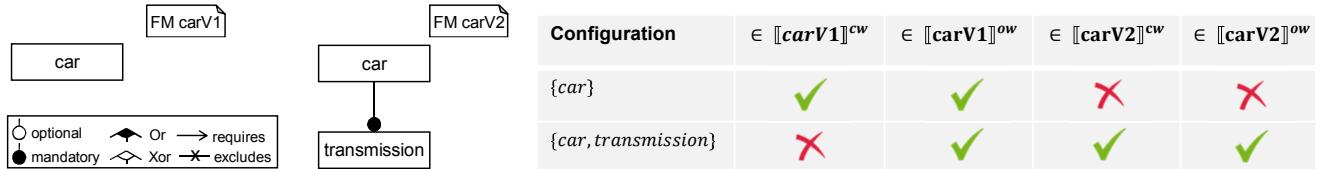


Figure 5: Simple feature models possible in early development stages.

If one FM contains at least all of the modeling elements that another FM contains, then the former FM's open-world semantics is a subset of the latter FM's open-world semantics. Especially in early development stages, this is a highly desired property as the addition of information (requirements) should only restrict the set of possible realizations, *i.e.*, valid configurations. This property does not necessarily hold for the closed-world semantics as illustrated in Figure 5. Every modeling element of the FM carV1 is also present in the FM carV2, but the closed-world semantics of carV2 is not subset of the closed-world semantics of carV1. On the other hand, the open-world semantics of carV2 is a subset of the open-world semantics of carV1. Hence, carV2 is an open-world but no closed-world refinement of carV1.

#### 4 SEMANTIC DIFFERENCING OF FEATURE MODELS FOR EVOLUTION ANALYSES

Model evolution analysis is an important task to detect bugs and explore design alternatives. Syntactic differencing approaches [3] are not concerned with the semantics of modeling languages. They reveal the syntactic elements that have changed between two successor model versions. The syntactic difference from one FM to another FM reveals syntactic changes that yield the latter FM when they are applied to the former FM [3]. Thus, the result of a syntactic differencing approach for FMs is independent of the semantics under consideration. In contrast, semantic differencing [2, 26, 35] abstracts from the syntax of models. The semantic difference from one model to another model contains the elements in the semantics of the former model that are not elements in the semantics of the latter model. Thus, semantic differencing reveals the models' differences by elements of their meanings [35]. The semantic difference from one FM to another FM is the set of elements of the one FM's semantics that are no members of the other FM's semantics:

**DEFINITION 5 (SEMANTIC DIFFERENCE).** Let  $FM_1$  and  $FM_2$  be two feature models.

The closed-world semantic difference from  $FM_1$  to  $FM_2$  is defined as  $\delta^{\text{cw}}(FM_1, FM_2) \stackrel{\text{def}}{=} \llbracket FM_1 \rrbracket^{\text{cw}} \setminus \llbracket FM_2 \rrbracket^{\text{cw}}$ .

The open-world semantic difference from  $FM_1$  to  $FM_2$  is defined as  $\delta^{\text{ow}}(FM_1, FM_2) \stackrel{\text{def}}{=} \llbracket FM_1 \rrbracket^{\text{ow}} \setminus \llbracket FM_2 \rrbracket^{\text{ow}}$ .

Depending on the semantics under consideration, semantic differencing may yield different results. Previous work produced a semantic differencing operator for FMs using the closed-world semantics [2, 11] to decide whether  $\delta^{\text{cw}}(FM_1, FM_2) = \emptyset$  for two FMs  $FM_1$  and  $FM_2$ . This operator is especially useful when analyzing the semantic differences between two FMs in late development stages.

By then, it is assumed that all possible features of the domain of interest are identified and contained in the FM modeling the product line. However, in early development stages, such as a specification phase in classical development processes, or when applying agile methods, requirements are subject to change. This also affects the set of possible features available in a product line during product line engineering, *i.e.*, the set of all features of the domain of interest is usually not known *a priori*. In this case, product line developers have an "open-world" view on the semantics of the product line under development - all features that are not explicitly constrained, are possible. In contrast, when using the closed-world semantics, all features that are not explicitly constrained, are not possible. For this reason, the existing operator for semantic differencing of FMs [2] using the closed-world semantics does not always yield intuitive results when used in early development stages.

For example, consider a car manufacturer starting with the very underspecified FM carV1 as depicted on the left hand side of Figure 5. This FM contains the single root feature **car**. Later in the development process, the manufacturer identifies that different car configurations support different types of transmission systems. The manufacturer thus changes the FM to carV2 as depicted on the right hand side of Figure 5. The manufacturer is still in the specification phase as the set of all possible car features is still to be identified. Therefore, as information is added to the model and no information is removed, the manufacturer would consider the FM carV2 to be a refinement of the FM carV1, *i.e.*, the manufacturer would expect that the semantics of carV2 is a subset of the semantics of carV1. This is the case when using the open-world semantics, *i.e.*,  $\delta^{\text{ow}}(\text{carV2}, \text{carV1}) = \emptyset$ . In contrast, the FM carV2 is no refinement of its predecessor version carV1 when using the closed-world semantics, because  $\{car, transmission\} \in \delta^{\text{cw}}(\text{carV2}, \text{carV1})$ . On the other hand, in late development stages, when all possible features are known, the closed-world semantics is useful for detecting whether new products have been explicitly added or removed - which might be a circumstance that remains undetected when using the open-world semantics. Thus, depending on the development stage, both semantics provide meaningful results for semantic differencing.

In general, it is not possible to conclude properties of the open-world semantic difference between two FMs based on the closed-world semantic difference between the same FMs, and vice versa: If one FM is a refinement of another FM using the closed-world semantics, then the latter FM is not necessarily a refinement of the former FM using the open-world semantics, and vice versa. Figure 6 illustrates this: The FMs  $FM_1$  and  $FM_2$  have the same open-world semantics. However, neither of the two FMs is a refinement of the

other FM using the closed-world semantics. Vice versa, the FM  $FM2$  is a refinement of the FM  $FM3$  using the closed-world semantics. However, under the open-world semantics the situation is reversed, i.e.,  $FM3$  is a refinement of  $FM2$  under the open-world semantics. However, if the two FMs under analysis share exactly the same features, then the two semantics are equivalent up to semantic differences, i.e., the closed-world semantic difference from the one FM to the other FM is empty if, and only if, the open-world semantic difference from the FM to the other FM is empty.

**LEMMA 5.** Let  $FM_1 = (F_1, E_1, r_1, child_1, I_1, EX_1)$  and  $FM_2 = (F_2, E_2, r_2, child_2, I_2, EX_2)$  be two feature models with  $F_1 = F_2$ . Then,  $\delta^{cw}(FM_1, FM_2) = \emptyset$  iff  $\delta^{ow}(FM_1, FM_2) = \emptyset$ .

**PROOF.** Let  $FM_1$  and  $FM_2$  be given as above.

" $\Rightarrow$ ": Assume  $\delta^{cw}(FM_1, FM_2) = \emptyset$ . Let  $c \in \llbracket FM_1 \rrbracket^{cw}$  be a configuration that is valid in  $FM_1$ . Using Lemma 1, we obtain that  $c \cap F_1$  is also valid in  $FM_1$ . As  $c \cap F_1$  is valid in  $FM_1$  and  $c \cap F_1 \subseteq F_1$ , it follows that  $c \cap F_1 \in \llbracket FM_1 \rrbracket^{cw}$  is a product of  $FM_1$ . As by assumption  $\delta^{cw}(FM_1, FM_2) = \emptyset$ , we obtain that  $c \cap F_1 \in \llbracket FM_2 \rrbracket^{cw}$  is also a product of  $FM_2$ . Therefore, as  $F_1 = F_2$ ,  $c \cap F_2 = c \cap F_1 \in \llbracket FM_2 \rrbracket^{ow}$  is valid in  $FM_2$ . Using Lemma 2, we obtain that  $c = (c \cap F_2) \cup (c \setminus F_2) \in \llbracket FM_2 \rrbracket^{ow}$  is also valid in  $FM_2$ . From the above, we can conclude that every valid configuration of  $FM_1$  is also a valid configuration of  $FM_2$  and, thus,  $\delta^{ow}(FM_1, FM_2) = \emptyset$ .

" $\Leftarrow$ ": Assume  $\delta^{ow}(FM_1, FM_2) = \emptyset$ . Let  $c \in \llbracket FM_1 \rrbracket^{cw}$  be a product of  $FM_1$ . Then, by definition  $c \in \llbracket FM_1 \rrbracket^{ow}$  is also valid in  $FM_1$ . As by assumption  $\delta^{ow}(FM_1, FM_2) = \emptyset$ , this implies  $c \in \llbracket FM_2 \rrbracket^{ow}$ . Therefore,  $c$  is valid in  $FM_2$ . As further,  $c \in \llbracket FM_1 \rrbracket^{cw}$ , it holds that  $c \subseteq F_1 = F_2$ . As  $c \subseteq F_2$  and  $c$  is valid in  $FM_2$ , we can conclude that  $c \in \llbracket FM_2 \rrbracket^{cw}$ . From the above, we can conclude that every product of  $FM_1$  is also a product of  $FM_2$  and, thus,  $\delta^{cw}(FM_1, FM_2) = \emptyset$ .  $\square$

The following introduces a semantic differencing operator for FMs using the open-world semantics. The operator is an automatic method for checking whether  $\delta^{ow}(FM_1, FM_2) = \emptyset$  for any two FMs  $FM_1$  and  $FM_2$ . It further yields a witness  $w \in \delta^{ow}(FM_1, FM_2)$ , if  $\delta^{ow}(FM_1, FM_2) = \emptyset$  does not hold. The enabler for the open-world semantic differencing method is that it suffices to search a finite set of configurations for a configuration that is valid in the one FM and not valid in the other FM. More specifically, it suffices to search all possible configurations that solely contain features that exist in the input FMs:

**THEOREM 1.** Let  $FM_1 = (F_1, E_1, r_1, child_1, I_1, EX_1)$  and  $FM_2 = (F_2, E_2, r_2, child_2, I_2, EX_2)$  be two feature models. Then,  $\llbracket FM_1 \rrbracket^{ow} \subseteq \llbracket FM_2 \rrbracket^{ow}$  iff  $(\llbracket FM_1 \rrbracket^{ow} \cap \wp(F_1 \cup F_2)) \subseteq (\llbracket FM_2 \rrbracket^{ow} \cap \wp(F_1 \cup F_2))$ .

**PROOF.** Let  $FM_1$  and  $FM_2$  be given as above.

" $\Rightarrow$ ": Assume  $\llbracket FM_1 \rrbracket^{ow} \subseteq \llbracket FM_2 \rrbracket^{ow}$  holds. This directly implies  $(\llbracket FM_1 \rrbracket^{ow} \cap \wp(F_1 \cup F_2)) \subseteq (\llbracket FM_2 \rrbracket^{ow} \cap \wp(F_1 \cup F_2))$ .

" $\Leftarrow$ ": Assume  $(\llbracket FM_1 \rrbracket^{ow} \cap \wp(F_1 \cup F_2)) \subseteq (\llbracket FM_2 \rrbracket^{ow} \cap \wp(F_1 \cup F_2))$  holds. Let  $C \in \llbracket FM_1 \rrbracket^{ow}$  be an arbitrary configuration that is valid in  $FM_1$ . We define  $C' \stackrel{\text{def}}{=} C \cap (F_1 \cup F_2)$ . As  $C \in \llbracket FM_1 \rrbracket^{ow}$  and  $C' \cap F_1 = (C \cap (F_1 \cup F_2)) \cap F_1 = C \cap F_1$ , using Lemma 1, it is guaranteed that  $C' \in \llbracket FM_1 \rrbracket^{ow}$ . Therefore, as  $C' \subseteq F_1 \cup F_2$ , we have that  $C' \in \llbracket FM_1 \rrbracket^{ow} \cap \wp(F_1 \cup F_2)$ . As by assumption  $(\llbracket FM_1 \rrbracket^{ow} \cap \wp(F_1 \cup F_2)) \subseteq (\llbracket FM_2 \rrbracket^{ow} \cap \wp(F_1 \cup F_2))$ , we obtain that  $C' \in \llbracket FM_2 \rrbracket^{ow} \cap \wp(F_1 \cup F_2)$ . We observe that  $C \setminus C' = C \setminus (C \cap (F_1 \cup F_2)) = C \setminus (F_1 \cup F_2) \subseteq U_F \setminus (F_1 \cup F_2)$ .

$F_2)$   $\subseteq U_F \setminus F_2$ . As also  $C' \in \llbracket FM_2 \rrbracket^{ow} \cap \wp(F_1 \cup F_2) \subseteq \llbracket FM_2 \rrbracket^{ow}$ , Lemma 2 guarantees that  $C' \cup (C \setminus C') \in \llbracket FM_2 \rrbracket^{ow}$ . Observing that  $C' \cup (C \setminus C') = C'$  and  $C' \cup C = C$  because  $C' \subseteq C$  by construction of  $C'$ , we can conclude  $C \in \llbracket FM_2 \rrbracket^{ow}$ .  $\square$

The above enables the definition of a simple algorithm for checking whether the open-world semantics of a FM  $FM_1$  is included in the open-world semantics of a FM  $FM_2$  based iteratively enumerating all configurations of features that are used in  $FM_1$  or  $FM_2$ , which are finitely many. Theorem 1 guarantees that there exist semantic differences from  $FM_1$  to  $FM_2$  under the open-world semantics if, and only if, one of these configurations is valid in  $FM_1$  and not valid in  $FM_2$ . More efficient implementations that exploit the optimizations implemented in modern SAT-solvers are also possible: Similar to the method for semantic FM differencing under the closed-world semantics [2], for two FMs  $FM_1$  and  $FM_2$ , we can construct two formulas  $\Phi_1$  and  $\Phi_2$  with the following properties: The formula  $\Phi_1$  contains a variable for each feature in  $FM_1$ . The formula  $\Phi_2$  contains a variable for each feature in  $FM_2$ . Each model of  $\Phi_1$  encodes a valid configuration of  $FM_1$  and each model of  $\Phi_2$  encodes a valid configuration of  $FM_2$ . If each feature that is used in both FMs is encoded by the same variable in both formulas, then each model of the formula  $\Phi_1 \wedge \neg \Phi_2$  encodes a valid configuration of  $FM_1$  that is no valid configuration of  $FM_2$ . Theorem 1 guarantees that there are semantic differences from  $FM_1$  to  $FM_2$  under the open-world semantics if, and only if,  $\Phi_1 \wedge \neg \Phi_2$  is satisfiable. Similarly, semantic differencing of the FMs under the closed-world semantics is possible via checking whether the following formula is satisfiable [2]:  $(\Phi_1 \wedge (\bigwedge_{f \in F_2 \setminus F_1} \neg x_f)) \wedge \neg(\Phi_2 \wedge (\bigwedge_{f \in F_1 \setminus F_2} \neg x_f))$  where  $F_1$  is the set of features of  $FM_1$ ,  $F_2$  is the set of features of  $FM_2$ , and  $x_f$  is the variable for feature  $f$ .

## 5 DISCUSSION: SEMANTICS IN DIFFERENT DEVELOPMENT PHASES

Feature oriented development processes [9, 25] divide into two phases, i.e., *domain engineering* in which “commonality and variability of the product line are defined and realised” [9], and *application engineering* “in which the applications of the product line are built by reusing domain artifacts and exploiting the product line variability” [9]. Semantic differencing of FMs facilitates to analyze how the range of products in the product line evolves. Independent of the semantics, incomparable semantics of two consecutive versions of a FM indicate that the new version excludes configurations from the original version and also adds configurations that were not possible in the original version.

During the domain engineering phase, features and their relationships are identified and added to FMs for obtaining a complete model of the product line. Open-world semantic differencing facilitates to compare consecutive versions of the FMs during domain engineering following the intuition that adding features actually refines the set of configurations described by the FM (cf. Section 4). During the *domain engineering phase*, product line specifications are created, i.e., features and their relationships are identified and added to a FM. Therefore, this phase usually underlies the assumption that the FM under development does not already contain all relevant features. Therefore, in this phase, the addition of information is usually considered to be a refinement in the sense that the addition



**Figure 6: Refinement using one of the semantics does not imply refinement using the other semantics.**

of information excludes elements from the model's semantics. Thus, using the open-world semantics usually yields the expected results for semantic differencing in the domain engineering phase.

During the *application engineering phase*, developers *exploit* (use) the FM created during the domain engineering phase. The individual features are realized and choosing a product of the FM composes the realizations of the product's features to obtain a realization of the product. In this phase, changing a FM such that it permits new products, e.g., by adding features, effectively changes the realizations of the product line's products and might even cause already implemented products to become invalid. Each newly added product should be detected and reviewed by product line engineers to ensure that the product line only permits meaningful realizations. Thus, in the application engineering phase, semantic differencing should report every newly added product when the set of products of a FM changes, e.g., when features are added. Therefore, semantic differencing using the closed-world semantics [2] yields the appropriate results in this development phase. Refinement of FMs in the application engineering phase corresponds to further restricting the already existing product range by excluding redundant products or products that are technically impossible to implement. Therefore, for ensuring that products are not removed unintentionally during the application engineering phase, the semantic difference from an original FM version to a successor version should detect all products that have been removed during the evolution to the successor version. This is also achievable with the closed-world semantics.

Semantic differencing is a valuable operation in software product line development, where the expected result depends on the semantics definition. Using the open-world semantics in early and the closed-world semantics in late development phases usually meets the purposes of the different development phases and the intuition of developers. Therefore, semantic FM evolution analysis by means of semantic differencing usually requires to adapt the semantics definition according to the development phase.

## 6 RELATED AND AFFECTED WORK

Existing approaches for semantic FM differencing [2] are based on the closed-world semantics. As discussed in Section 4 and Section 3, using the open-world semantics may change the result when conducting semantic differencing significantly. The appropriate semantics used for semantic differencing depends on the analyst's intuition, which usually differs in early and late development phases (cf. Section 5). Semantic differencing is only one out of many automated analyses for FMs (e.g., [6, 42, 47, 53]). To the best of our knowledge, most semantic analyses of FMs are based on the closed-world semantics. In general, all of these analyses are also applicable

using the open-world semantics. However, using the open-world semantics might change the perspective of the analyses in the sense that they provide other information as the analyses using the closed-world semantics.

This section presents related work on automated FM analyses, composition, and synthesis. For each of the approaches, we discuss the differences from the existing method's results under the closed-world semantics to the results under the open-world semantics.

### 6.1 Translations for Automated Reasoning

There are well-known translations from FMs to propositional formulas such that the interpretations satisfying the formula obtained from translating a FM represent exactly the products of the FM [2, 6, 8, 16, 55]. As presented in Section 4, these translations are easily adaptable for semantic FM differencing using the open-world semantics. Similarly, all reasoning approaches on FMs based on propositional logic using, e.g., binary decision diagrams [10] as proposed in [7], can be applied for open-world semantic analyses by adapting the translation to propositional logic. Description logic is another formalism used for automated reasoning on FMs [6, 54]. The approach [54] assumes closed-world semantics and can be similarly adapted as the approaches using propositional formulas to base the analyses on the open-world semantics. Furthermore, automatic reasoning on FMs has been conducted using constraint programming [50]. The rules for mapping a FM to a constraint satisfaction problem [8] can equally be adapted by excluding the constraints, which state that features have to be contained in the FM's set of features.

### 6.2 Semantic Property Analyses

There are automatic analyses for the plausibility of FMs in terms of properties of their semantics. As these analyses are based on the semantics of FMs, the analysis results change when using the open-world semantics instead of the closed-world semantics.

*Dead Feature Detection.* A feature is dead in a FM, if it is not part of any of the FM's modeled configurations [6]. With the closed-world semantics, a feature is dead if it is not modeled in the FM or if it is constrained by a cross-tree constraints such that it cannot be part of any modeled configuration [6]. Under the open-world semantics, a feature not modeled in a FM is not dead. In case the feature is modeled in the FM, it is dead under the closed-world semantics, if it is dead under the open-world semantics.

*Verifying Products or Partial Configurations.* Operators for verifying products check whether a product is valid in the modeled

product line [6]. As defined in Section 3, every product of a FM (element of the closed-world semantics) is also valid in the FM (element of the open-world semantics). Choosing the appropriate semantics depends on the intuition of the product line engineer when using the operator (cf. Section 5). A similar operation is the verification of partial configurations [6]. Valid partial configurations must not contain contradictions with respect to the cross-tree constraints modeled in the FM. Again, the result of applying the operator depends on the chosen semantics and the appropriate semantics depends on the developer’s intuition as well as the current development phase.

*Feature Model Satisfiability.* A FM is satisfiable, if it permits at least one configuration, *i.e.*, its semantics contains at least one element that is different from the empty configuration [6]. If a FM is not satisfiable, then it contains contradicting cross-tree constraints [6]. There are well-known automated methods for checking whether a FM is satisfiable under the closed-world semantics [5, 6, 24, 48], which are, *e.g.*, based on checking the satisfiability of the formula resulting from translating the FM to propositional logic. The approaches are reusable for consistency checking using the open-world semantics, if the universe of features is finite and the propositional formula resulting from the translation is interpreted over the variables encoded by the elements of the finite universe. If the universe of features is infinite, then every FM is satisfiable under the open-world semantics (cf. Lemma 4). This reflects the suitability of the open-world semantics in the domain engineering phase, where feature constraints are yet to be identified.

*Corrective Explanations.* A corrective explanation provides information that explain why an operator yields its result. This facilitates developers in finding deficiencies that should be corrected. Examples for corrective explanations have been proposed, *e.g.*, by [49] introducing abductive reasoning or by [11, 26, 34] for analyzing the syntactic changes, which lead to some semantic relation (such as refinement) between two FMs. For instance, [26] presents a language independent theory to obtain a sequence of change operations to repair a failed model refinement. The formal approach is exemplary applied to FMs using the closed-world semantics. Each of the approaches [11, 26, 34, 49] bases its semantic FM analyses on the propositional formulas obtained from translating FMs. The analyses of the approaches are adaptable to the open-world semantics by interchanging the translation from FMs to propositional formulas as discussed above.

*Product enumeration methods.* Operators to determine the number of specified products [8, 15] or even the entire range of possible products aim at revealing the level of variability within the product line and to identify possible extensions of the initially intended product scope [6]. Feature dependency analyses [39] are used to identify undesirable dependencies between features within a FM. Using the open-world semantics, the results of these analyzes are usually (assuming an infinite universe of features) infinite sets that do not provide valuable information. Furthermore, these operators analyze the variability or deficiencies within a modeled product line that is assumed to be almost complete. Therefore, applying these analyses is not meaningful in early development phases, where many features of the domain of interest may be missing in the FM.

### 6.3 Semantics-aware Feature Model Composition and Synthesis

Composition and synthesis of FMs are valuable operations that target FM reuse in software product line development.

*Feature Model Composition.* There are various composition operators for FMs [1, 2, 43, 51, 52]. Each composition operator combines two FMs to obtain another FM that satisfies certain properties with respect to the original FMs’ semantics. For instance, an intersection composition operator takes two FMs as input and returns another FM with a semantics that is the intersection or a superset of the intersection of the semantics of the input FMs [1, 2, 43, 51]. Analogously, a FM union composition operator takes two FMs as input and returns a new FM with a semantics that is the union or a superset of the union of the input FMs’ semantics [1, 2, 43, 52]. The existing composition operators consider the closed-world semantics. The relationship between the existing composition operators and the open-world semantics reveals reuse potential: Assume reusing an intersection composition operator that composes two FMs that share the same features such that the resulting FM has the same features as the input FMs and the resulting FM’s closed-world semantics is equal to the intersection of the input FMs’ closed-world semantics. Then, the resulting FM’s open-world semantics is equal to the intersection of the input FMs’ open-world semantics.

**LEMMA 6.** *Let  $FM_1, FM_2, FM_3$  be feature models that all share the same set of features. If  $\llbracket FM_3 \rrbracket^{cw} = \llbracket FM_1 \rrbracket^{cw} \cap \llbracket FM_2 \rrbracket^{cw}$ , then  $\llbracket FM_3 \rrbracket^{ow} = \llbracket FM_1 \rrbracket^{ow} \cap \llbracket FM_2 \rrbracket^{ow}$ .*

**PROOF.** Let  $FM_1, FM_2, FM_3$  be three feature models sharing the same features such that  $\llbracket FM_3 \rrbracket^{cw} = \llbracket FM_1 \rrbracket^{cw} \cap \llbracket FM_2 \rrbracket^{cw}$ . Let  $F$  denote the set of features of the three FMs.

“ $\subseteq$ ”: Let  $c \in \llbracket FM_3 \rrbracket^{ow}$  be a valid configuration of  $FM_3$ . By Lemma 1, we have that  $c \cap F \in \llbracket FM_3 \rrbracket^{ow}$  is also valid in  $FM_3$ . As  $c \cap F$  is valid in  $FM_3$  and  $(c \cap F) \subseteq F$ , we obtain with Definition 4 that  $c \cap F \in \llbracket FM_3 \rrbracket^{cw}$ . From this, as by assumption it holds that  $\llbracket FM_3 \rrbracket^{cw} = \llbracket FM_1 \rrbracket^{cw} \cap \llbracket FM_2 \rrbracket^{cw}$ , we can infer that  $c \cap F \in \llbracket FM_1 \rrbracket^{cw}$  and  $c \cap F \in \llbracket FM_2 \rrbracket^{cw}$ . Therefore, with Lemma 3, we obtain  $c \cap F \in \llbracket FM_1 \rrbracket^{ow}$  and  $c \cap F \in \llbracket FM_2 \rrbracket^{ow}$ . With this and as  $c \setminus F \subseteq U_F \setminus F$ , we can conclude with Lemma 2 that  $c = (c \cap F) \cup (c \setminus F) \in \llbracket FM_1 \rrbracket^{ow}$  and  $c = (c \cap F) \cup (c \setminus F) \in \llbracket FM_2 \rrbracket^{ow}$ . Therefore,  $c \in \llbracket FM_1 \rrbracket^{ow} \cap \llbracket FM_2 \rrbracket^{ow}$ .

“ $\supseteq$ ”: Let  $c \in \llbracket FM_1 \rrbracket^{ow} \cap \llbracket FM_2 \rrbracket^{ow}$  be a configuration that is valid in  $FM_1$  and in  $FM_2$ . By Lemma 1, we have that  $c \cap F \in \llbracket FM_1 \rrbracket^{ow} \cap \llbracket FM_2 \rrbracket^{ow}$  is also valid in  $FM_1$  and in  $FM_2$ . As  $c \cap F$  is valid in  $FM_1$  and in  $FM_2$  and as  $c \cap F \subseteq F$ , we obtain with Definition 4 that  $c \cap F \in \llbracket FM_1 \rrbracket^{cw} \cap \llbracket FM_2 \rrbracket^{cw}$ . Using the assumption  $\llbracket FM_3 \rrbracket^{cw} = \llbracket FM_1 \rrbracket^{cw} \cap \llbracket FM_2 \rrbracket^{cw}$ , we obtain that  $c \cap F \in \llbracket FM_3 \rrbracket^{cw}$ . With Lemma 3, this implies  $c \cap F \in \llbracket FM_3 \rrbracket^{ow}$ . With this and as  $c \setminus F \subseteq U_F \setminus F$ , we can conclude with Lemma 2 that  $c = (c \cap F) \cup (c \setminus F) \in \llbracket FM_3 \rrbracket^{ow}$ .  $\square$

Reusing union composition operators yields similar results, *i.e.*, the resulting FM’s open-world semantics is equal to the union of the input FMs’ open-world semantics.

**LEMMA 7.** *Let  $FM_1, FM_2, FM_3$  be feature models that all share the same set of features. If  $\llbracket FM_3 \rrbracket^{cw} = \llbracket FM_1 \rrbracket^{cw} \cup \llbracket FM_2 \rrbracket^{cw}$ , then  $\llbracket FM_3 \rrbracket^{ow} = \llbracket FM_1 \rrbracket^{ow} \cup \llbracket FM_2 \rrbracket^{ow}$ .*

**PROOF.** Let  $FM_1, FM_2, FM_3$  be three feature models sharing the same features such that  $\llbracket FM_3 \rrbracket^{cw} = \llbracket FM_1 \rrbracket^{cw} \cup \llbracket FM_2 \rrbracket^{cw}$ . Let  $F$  denote the set of features of the three FMs.

" $\subseteq$ ": Let  $c \in \llbracket FM_3 \rrbracket^{ow}$  be a valid configuration of  $FM_3$ . By Lemma 1, we have that  $c \cap F \in \llbracket FM_3 \rrbracket^{ow}$  is also valid in  $FM_3$ . As  $c \cap F$  is valid in  $FM_3$  and  $(c \cap F) \subseteq F$ , we obtain with Definition 4 that  $c \cap F \in \llbracket FM_3 \rrbracket^{cw}$ . From this, as by assumption it holds that  $\llbracket FM_3 \rrbracket^{cw} = \llbracket FM_1 \rrbracket^{cw} \cup \llbracket FM_2 \rrbracket^{cw}$ , we can infer that  $c \cap F \in \llbracket FM_1 \rrbracket^{cw}$  or  $c \cap F \in \llbracket FM_2 \rrbracket^{cw}$ . Therefore, with Lemma 3, we obtain  $c \cap F \in \llbracket FM_1 \rrbracket^{ow}$  or  $c \cap F \in \llbracket FM_2 \rrbracket^{ow}$ . With this and as  $c \setminus F \subseteq UF \setminus F$ , we can conclude with Lemma 2 that  $c = (c \cap F) \cup (c \setminus F) \in \llbracket FM_1 \rrbracket^{ow}$  or  $c = (c \cap F) \cup (c \setminus F) \in \llbracket FM_2 \rrbracket^{ow}$ . Therefore,  $c \in \llbracket FM_1 \rrbracket^{ow} \cup \llbracket FM_2 \rrbracket^{ow}$ .

" $\supseteq$ ": Let  $c \in \llbracket FM_1 \rrbracket^{ow} \cup \llbracket FM_2 \rrbracket^{ow}$  be a configuration that is valid in  $FM_1$  or in  $FM_2$ . By Lemma 1, we have that  $c \cap F \in \llbracket FM_1 \rrbracket^{ow} \cup \llbracket FM_2 \rrbracket^{ow}$  is also valid in  $FM_1$  or in  $FM_2$ . As  $c \cap F$  is valid in  $FM_1$  or in  $FM_2$  and as  $c \cap F \subseteq F$ , we obtain with Definition 4 that  $c \cap F \in \llbracket FM_1 \rrbracket^{cw} \cup \llbracket FM_2 \rrbracket^{cw}$ . Using the assumption  $\llbracket FM_3 \rrbracket^{cw} = \llbracket FM_1 \rrbracket^{cw} \cup \llbracket FM_2 \rrbracket^{cw}$ , we obtain that  $c \cap F \in \llbracket FM_3 \rrbracket^{cw}$ . With Lemma 3, this implies  $c \cap F \in \llbracket FM_3 \rrbracket^{ow}$ . With this and as  $c \setminus F \subseteq UF \setminus F$ , we can conclude with Lemma 2 that  $c = (c \cap F) \cup (c \setminus F) \in \llbracket FM_3 \rrbracket^{ow}$ .  $\square$

**Feature Model Synthesis.** The FM synthesis problem [16] takes a propositional formula over features as input. The idea is to compute a FM such that its closed-world semantics is equal to the satisfying interpretations of the formula. Concerning the open-world semantics, the translation constructs a FM such that all the constraints induced by the formula must be respected by the FM's valid configurations. This also implies that every configuration over the FM's features that is no product of the resulting FM is also no valid configuration of the resulting FM. Existing FM synthesis methods are thus well reusable when using the open-world semantics.

## 7 CONCLUSION AND FUTURE PROSPECTS

Semantic differencing of FMs facilitates to analyze how the range of products in the product line evolves. Using approaches for semantic differencing based on the open-world semantics, additionally to those based on the closed-world semantics, supports model evolution analysis of product refinement, especially in the design phase. Our semantic differencing operator for FMs using the open-world semantics facilitates the comparison of FMs for product line managers during early development stages. The proposed open-world semantics is especially useful in early development stages or when using agile development, where requirements are permanently subject to change. Automatic semantic differencing of FMs using the open-world semantics is possible as it suffices to search all possible configurations that solely contain features that exist in the input FMs. As argued in Section 5, using both, open-world semantics in early development stages, and closed-world semantics in late development stages, provides developers the best of both worlds. For the early development stages, it is important that the addition of information to a model causes that the resulting model is a refinement of its predecessor version.

The integration of the semantics in various tools is subject to future work. For developers, a graphical representation of valid configurations in the compared FMs could be useful to facilitate the

understanding of the differences. Moreover, it will be interesting to integrate the implementation of semantic FM differencing in a tool, which utilizes the different approaches for syntactic and semantic differencing of FMs to provide a comfortable way to analyze FMs.

## REFERENCES

- [1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. 2010. Composing Feature Models. In *Software Language Engineering*.
- [2] Mathieu Acher, Patrick Heymans, Philippe Lahire, Clément Quinton, Philippe Collet, and Philippe Merle. 2012. Feature Model Differences. In *Advanced Information Systems Engineering - 24th International Conference*.
- [3] Marcus Alelan and Ivan Porres. 2003. Difference and Union of Models. In «UML» 2003 - The Unified Modeling Language. *Modeling Languages and Applications*.
- [4] Sven Apel and Christian Kästner. 2009. An Overview of Feature-Oriented Software Development. *Journal of Object Technology* 8, 5 (2009), 49–84.
- [5] Dor Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Software Product Lines*.
- [6] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–636.
- [7] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2007. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*.
- [8] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*.
- [9] Böckle, Günter and Pohl, Klaus and van der Linden, Frank. 2005. A Framework for Software Product Line Engineering. In *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer Berlin Heidelberg, 19–38.
- [10] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* 35, 8 (1986), 677–691.
- [11] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. 2016. Reasoning About Product-line Evolution Using Complex Feature Model Differences. *Automated Software Engineering* 23, 4 (2016), 687–733.
- [12] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2017. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture*.
- [13] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés, and Mike Hinckley. 2014. An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. *Journal of Systems and Software* 91 (2014).
- [14] Paul Clements and Linda Northrop. 2007. *Software Product Lines: Practices and Patterns*. Addison-Wesley.
- [15] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing Cardinality-based Feature Models and their Specialization. In *Software Process: Improvement and Practice*, Vol. 10, 7–29.
- [16] Krzysztof Czarnecki and Andrzej Wasowski. 2007. Feature Diagrams and Logics: There and Back Again. In *Software Product Line Conference (SPLC 2007)*.
- [17] Ferruccio Damiani, Luca Padovani, and Ina Schaefer. 2012. A Formal Foundation for Dynamic Delta-oriented Software Product Lines. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*.
- [18] Uli Fahrenberg, Mathieu Acher, Axel Legay, and Andrzej Wasowski. 2014. Sound Merging and Differencing for Class Diagrams. In *Fundamental Approaches to Software Engineering*.
- [19] Uli Fahrenberg, Axel Legay, and Andrzej Wasowski. 2011. Vision Paper: Make a Difference! (Semantically). In *Model Driven Engineering Languages and Systems*.
- [20] Robert France and Bernhard Rumpe. 2007. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE*.
- [21] Christian Gerth, Jochen M. Küster, Markus Luckey, and Gregor Engels. 2010. Precise Detection of Conflicting Change Operations Using Process Model Terms. In *Model Driven Engineering Languages and Systems*.
- [22] Hans Gröniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. 2008. Modeling Variants of Automotive Systems using Views. In *Proceedings of Workshop Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (MBEFF)*.
- [23] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. 2007. An Algebraic View on the Semantics of Model Composition. In *Model Driven Architecture- Foundations and Applications*.
- [24] Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Software Engineering Institute - Carnegie Mellon University.
- [25] Kang, Kyo C. and Kim, Sajoong and Lee, Jaejoon and Kim, Kijoo and Shin, Euiseob and Huh, Moonhang. 1998. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering* 5, 1.

- [26] Oliver Kautz and Bernhard Rumpe. 2018. On Computing Instructions to Repair Failed Model Refinements. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*.
- [27] Philipp Kehrbusch, Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, and Christoph Schulze. 2016. Interface-based Similarity Analysis of Software Components for the Automotive Industry. In *International Systems and Software Product Line Conference (SPLC '16)*.
- [28] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2011. A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning. In *International Conference on Automated Software Engineering (ASE'11)*.
- [29] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2013. Consistency-Preserving Edit Scripts in Model Versioning. In *International Conference on Automated Software Engineering (ASE)*.
- [30] Jochen M. Küster, Christian Gerth, and Gregor Engels. 2009. Dependent and Conflicting Change Operations of Process Models. In *Model Driven Architecture - Foundations and Applications*.
- [31] Jochen M. Küster, Christian Gerth, Alexander Förster, and Gregor Engels. 2008. Detecting and Resolving Process Model Differences in the Absence of a Change Log. In *Business Process Management*.
- [32] Philip Langer, Tanja Mayerhofer, and Gerti Kappel. 2014. A Generic Framework for Realizing Semantic Model Differencing Operators. In *PSRC@MoDELS (CEUR Workshop Proceedings)*, Vol. 1258. CEUR-WS.org.
- [33] Philip Langer, Tanja Mayerhofer, and Gerti Kappel. 2014. Semantic Model Differencing Utilizing Behavioral Semantics Specifications. In *Model-Driven Engineering Languages and Systems*.
- [34] Shahar Maoz and Jan Oliver Ringert. 2016. A framework for relating syntactic and semantic model differences. *Software & System Modeling* 17, 3 (2016).
- [35] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2010. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10) (LNCS 6627)*. Springer, 194–203.
- [36] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, 179–189.
- [37] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CDDiff: Semantic Differencing for Class Diagrams. In *ECCOP 2011 – Object-Oriented Programming*.
- [38] Tanja Mayerhofer, Philip Langer, and Gerti Kappel. 2015. Semantic Model Differencing Based on Execution Traces. In *Software Engineering & Management*.
- [39] Marcílio Mendonça, Donald Cowan, William Mayyk, and Toacy Oliveira. 2008. Collaborative Product Configuration: Formalization and Efficient Algorithms for Dependency Analysis. *Journal of Software* 3, 2 (2008).
- [40] Klaus Pohl, Günther Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg.
- [41] Raymond Reiter. 1978. On Closed World Data Bases. In *Logic and Data Bases*. Springer US.
- [42] Camille Salinesi and Raúl Mazo. 2012. Defects in Product Line Models and how to Identify them. In *Software Product Line - Advanced Topic*. InTech editions.
- [43] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic Semantics of Feature Diagrams. *Computer Networks* 51, 2 (2007).
- [44] Mirjam Steger, Christian Tischer, Birgit Boss, Andreas Müller, Oliver Pertler, Wolfgang Stolz, and Stefan Ferber. 2004. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In *Software Product Line Conference (SPLC'04)*.
- [45] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. 2014. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software & Systems Modeling* 13, 1 (2014).
- [46] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning about Edits to Feature Models. In *Proceedings of the 31st International Conference on Software Engineering*.
- [47] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014).
- [48] Pablo Trinidad, David Benavides, Amador Durán, Antonio Ruiz-Cortés, and Miguel Toro. 2008. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software* 81, 6 (2008).
- [49] Pablo Trinidad and Antonio Ruiz Cortés. 2009. Abductive Reasoning and Automated Analysis of Feature Models: How are they connected?. In *Third International Workshop on Variability Modelling of Software-Intensive Systems*.
- [50] Edward Tsang. 1993. *Foundations of Constraint Satisfaction*. Academic Press.
- [51] Pim van den Broek. 2012. Intersection of Feature Models. In *Software Product Line Conference (SPLC'12)*.
- [52] Pim van den Broek, Ismênia Galvão, and Joost Noppen. 2010. Merging Feature Models. In *Software Product Line Conference (SPLC'10)*.
- [53] Thomas von der Maßen and Horst Lichter. 2004. Deficiencies in Feature Models. In *Workshop on Software Variability Management for Product Derivation*.
- [54] Hai Wang, Yuan Fang Li, Jing Sun, Hongyu Zhang, and Jeff Pan. 2005. A Semantic Web Approach to Feature Modeling and Verification. In *Workshop on Semantic Web Enabled Software Engineering (SWESE'05)*.
- [55] Wei Zhang, Haiyan Zhao, and Hong Mei. 2004. A Propositional Logic-Based Method for Verification of Feature Models. In *Formal Methods and Software Engineering (FSE'04)*. Springer Berlin Heidelberg.

# Achieving Change Requirements of Feature Models by an Evolutionary Approach

## Extended Abstract

Paolo Arcaini

National Institute of Informatics

Tokyo, Japan

arcaini@nii.ac.jp

Angelo Gargantini

University of Bergamo

Bergamo, Italy

angelo.gargantini@unibg.it

Marco Radavelli

University of Bergamo

Bergamo, Italy

marco.radavelli@unibg.it

## CCS CONCEPTS

- Software and its engineering → Software product lines; Search-based software engineering;
- Theory of computation → Evolutionary algorithms;

## KEYWORDS

feature models, software product lines, mutation, search-based software engineering

### ACM Reference Format:

Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. 2019. Achieving Change Requirements of Feature Models by an Evolutionary Approach: Extended Abstract. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342375>

## DESCRIPTION OF THE JOURNAL-FIRST PAPER

Software Product Lines (SPLs) are families of products that share some common features, and differ on some others. The variability of SPLs is usually described at design time by using *variability models*; one of the main used variability models are *feature models* (FMs).

Overtime, feature models need to be updated in order to avoid the risk of having a model with wrong features and/or wrong constraints. Two main causes for *change requirements* exist: either the model is incorrect (it excludes/includes some products that should be included/excluded), or the SPL has changed. The change requirements come from different sources: *failing tests* identifying configurations evaluated not correctly, or *business requirements* to add new products, to allow new features, to not support some products any more, and so on. All these change requirements identify configurations/features to add or remove, but do not identify a way to modify the feature model to achieve them. Manually updating a feature model to achieve all the change requirements can be particularly difficult and, in any case, error-prone and time consuming.

In the corresponding *journal-first paper* [1], we propose an approach (shown in Fig. 1) to automatically update a feature model upon change requirements. The user must only specify an *update request*, based on the change requirements coming from testing or from business requirements. The update request is composed of

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7138-4/19/09.

<https://doi.org/10.1145/3336294.3342375>

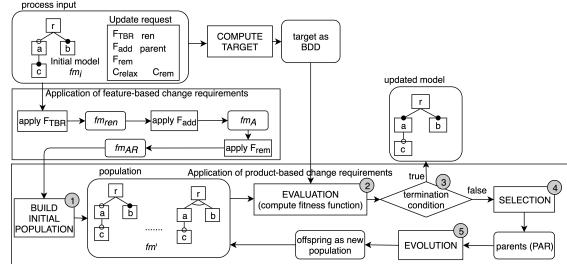


Figure 1: Proposed evolutionary approach

three kinds of *feature-based* change requirements and two kinds of *product-based* ones; the feature-based ones consist in features that must be renamed, and features that must be added to and removed from the products of the original feature model; the product-based ones, instead, consist in configurations that should be no more accepted as products by the final model, and configurations that should instead be accepted as new products. Starting from the update request, the approach tries to apply the feature-based change requirements directly on the starting model; however, some of these requirements could be not completely fulfilled. Then, by means of an evolutionary algorithm, the approach tries to obtain a feature model that captures all the change requirements: the feature-based ones not fulfilled in the previous phase, and the product-based ones. The process iteratively generates, from the current population of candidate solutions, a new population of feature models by mutation. Population members are evaluated considering primarily the percentage of correctly evaluated configurations, and secondly a measure of the *structural complexity* of the model, defined in terms of number of cross-tree constraints. When a correct model is found or some other termination condition holds, the process terminates returning as final model the one having the highest fitness value.

The approach has been evaluated on real-world feature models; experiments show that, on average, around 89% of requested changes are applied.

## ACKNOWLEDGMENTS

P. Arcaini is supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST. Funding Reference number: 10.13039/501100009024 ERATO.

## REFERENCES

- [1] Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. 2019. Achieving change requirements of feature models by an evolutionary approach. *Journal of Systems and Software* 150 (2019), 64–76. <https://doi.org/10.1016/j.jss.2019.01.045>

# Foundations of Collaborative, Real-Time Feature Modeling

Elias Kuiter  
Otto-von-Guericke-University  
Magdeburg, Germany  
kuiter@ovgu.de

Sebastian Krieter  
Harz University of Applied Sciences  
Otto-von-Guericke-University  
Wernigerode & Magdeburg, Germany  
skrieter@hs-harz.de

Jacob Krüger  
Otto-von-Guericke-University  
Magdeburg, Germany  
jkrueger@ovgu.de

Thomas Leich  
Harz University of Applied Sciences  
Wernigerode, Germany  
leich@hs-harz.de

Gunter Saake  
Otto-von-Guericke-University  
Magdeburg, Germany  
saake@ovgu.de

## ABSTRACT

Feature models are core artifacts in software-product-line engineering to manage, maintain, and configure variability. Feature modeling can be a cross-cutting concern that integrates technical and business aspects of a software system. Consequently, for large systems, a team of developers and other stakeholders may be involved in the modeling process. In such scenarios, it can be useful to utilize collaborative, real-time feature modeling, analogous to collaborative text editing in Google Docs or Overleaf. However, current techniques and tools only support a single developer to work on a model at a time. Collaborative and simultaneous editing of the same model is often achieved by using version control systems, which can cause merge conflicts and do not allow immediate verification of a model, hampering real-time collaboration outside of face-to-face meetings. In this paper, we describe the formal foundations of collaborative, real-time feature modeling, focusing on concurrency control by synchronizing multiple actions of collaborators in a distributed network. We further report on preliminary results, including an initial prototype. Our contribution provides the basis for extending feature-modeling tools to enable advanced collaborative feature modeling and integrate it with related tasks.

## CCS CONCEPTS

- Software and its engineering → Feature interaction; Software product lines; Programming teams.

## KEYWORDS

Software Product Line, Groupware, Feature Modeling, Consistency Maintenance, Collaboration

## ACM Reference Format:

Elias Kuiter, Sebastian Krieter, Jacob Krüger, Thomas Leich, and Gunter Saake. 2019. Foundations of Collaborative, Real-Time Feature Modeling. In *23rd International Systems and Software Product Line Conference - Volume A*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-7138-4/19/09...\$15.00  
<https://doi.org/10.1145/3336294.3336308>

(SPLC '19), September 9–13, 2019, Paris, France. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3336294.3336308>

## 1 INTRODUCTION

Variability modeling is a core activity for developing and managing a Software Product Line (SPL) [3, 19, 42]. It does not only concern implementation artifacts, but various other aspects of an SPL as well [7, 18]. For instance, a variability model often incorporates design decisions specific to an SPL's domain, but also provides a layer of abstraction that end users can comprehend. Thus, for large-scale projects, multiple people must work corporately to create a meaningful variability model [6, 38].

Numerous tools facilitate variability modeling with specialized user interfaces and automated model analyses (e.g., *FeatureIDE* [35], *pure::variants* [8], and *Gears* [28]). However, we are not aware of a technique for variability modeling that supports *collaborative, real-time editing*, similar to Google Docs or Overleaf, which hampers efficient cooperation during the modeling process. To the best of our knowledge, existing tools allow only a single user to edit a variability model at a time. Using version control systems, such as Git, developers can share and distribute variability models, but this is neither in real-time nor does it support a semantically meaningful resolution of conflicts. We see various potential use cases for collaborative, real-time variability modeling, for example:

- Multiple domain engineers can work simultaneously on the same variability model, either on different tasks (e.g., editing existing constraints) or on a coordinated task (e.g., introducing a set of new features).
- Engineers can share and discuss the variability model with domain experts, allowing to evolve it with real-time feedback without requiring costly co-location of the participants.
- Lecturers can teach variability-modeling concepts in a collaborative manner and can more easily involve the audience in hands-on exercises.

The most established form of variability models are *feature models* and their visual representations, *feature diagrams* [7, 16, 19]—on which we focus in this paper. Feature models capture the features of an SPL and their interdependencies, thereby defining the user-visible functionalities that are common or different for variants.

In this paper, we describe the conceptual foundations for our technique to achieve collaborative, real-time feature modeling. More precisely, we define the requirements our technique needs to fulfill

and, based on these requirements, derive a formal description of collaborative, real-time feature modeling that allows us to ensure its correct behavior and guides the actual implementation. This includes defining a basic set of feature-modeling operations, a conflict relation between these operations, and mechanisms for detecting and resolving potential conflicts. For this purpose, we extend existing techniques for real-time collaboration to provide the basis for our and future collaborative, real-time variability modeling techniques, which may be integrated into existing tools. Although we have implemented an initial prototype to demonstrate the feasibility of our technique, we do not elaborate on its technical details. Overall, we make the following contributions:

- We identify and describe what requirements should be fulfilled by a collaborative, real-time feature modeling technique and a corresponding editor.
- We define a concurrency control technique to allow for collaborative, real-time feature modeling. In particular, we introduce strategies and mechanisms to detect and resolve conflicts, thereby ensuring that the edited feature models remain syntactically and semantically consistent.
- We briefly report on preliminary results of our technique with regard to formal correctness and an initial prototype.

We aim to support uses cases that are based on three general conditions. First, we assume that users need or want to work simultaneously on the same feature model, for instance, to coordinate their efforts when performing independent or interacting tasks [33]. Thus, mechanisms for concurrency control are required. Second, we assume that a rather small team (i.e., no more than ten developers) is maintaining a feature model, based on studies on real-world SPLs [6, 24, 38]. For larger teams, managing collaborations and automatically resolving conflicts becomes much harder. Finally, we assume that not all developers work co-located, but are remotely connected [33], for which we aim to support both peer-to-peer and client-server architectures [44].

## 2 FORMAL FOUNDATIONS

Within this paper, we present a formal technique for collaborative, real-time feature modeling. In the following, we briefly introduce the notation of feature models and key concepts regarding consistency maintenance in collaborative editing systems.

### 2.1 Feature Modeling

We consider feature models in the form of feature diagrams, which specify the variability of SPLs using a hierarchy of features and cross-tree constraints. Thus, we define a feature model as follows:

**Definition 1.** A feature model  $FM$  is a tuple  $(\mathcal{F}, C)$  where  $\mathcal{F}$  is a finite set of features and  $C$  is a finite set of cross-tree constraints.<sup>1</sup>

A feature  $F \in FM.\mathcal{F}$  is a tuple  $(ID, parentID, optional, groupType, abstract, name)$  where  $ID \in ID$ ,  $parentID \in ID \cup \{\perp, \dagger\}$ ,  $optional \in \{true, false\}$ ,  $groupType \in \{and, or, alternative\}$ ,  $abstract \in \{true, false\}$ , and  $name$  is a string.

<sup>1</sup>For easier readability, we use a dot notation to access a tuple's elements (or *attributes*). For instance,  $FM.\mathcal{F}$  refers to the features in the feature model  $FM$ . Further, we interpret  $\mathcal{F}$  and  $C$  as functions to facilitate attribute lookup; e.g.,  $FM.\mathcal{F}(F^ID)$  refers to the feature (uniquely) identified by  $F^ID$  in  $FM$ .

A cross-tree constraint  $C \in FM.C$  is a tuple  $(ID, \phi)$  where  $ID \in ID$  and  $\phi$  is a propositional formula with variables ranging over  $ID$ , i.e.,  $Var(\phi) \subseteq ID$ .

The set  $ID$  contains all Universally Unique Identifiers (UUIDs) that can be used within a feature model.  $\perp$  denotes the *parentID* of the root feature, while  $\dagger$  denotes the *parentID* of an uninitialized feature.

To simplify our definitions, we declare the two sets  $\mathcal{F}_{FM}^{ID}$  and  $C_{FM}^{ID}$  for feature and constraint identifiers and the parent-child relation between features *descends from* ( $\leq_{FM}$ ) as follows:

**Definition 2.** Let  $FM$  be a feature model. Further, define

- $\mathcal{F}_{FM}^{ID} := \{F.ID \mid F \in FM.\mathcal{F}\}$ ,
- $C_{FM}^{ID} := \{C.ID \mid C \in FM.C\}$ , and
- $\leq_{FM}$  as the reflexive transitive closure of  $\{(A.ID, B^ID) \mid A \in FM.\mathcal{F}, B^ID \in \mathcal{F}_{FM}^{ID} \cup \{\perp, \dagger\} \wedge A.parentID = B^ID\}$ .

Using our notation from Definition 1, we can formally describe any feature model—using its feature diagram representation. However, this definition still allows that integral properties of feature models are violated. These properties are important, as we intend to manipulate feature models by means of operations. Thus, we also define the following conditions to describe *legal* feature models:

**Definition 3.** A feature model  $FM$  is considered legal iff all of the following conditions are true:

- Unique identifiers:  $|FM.\mathcal{F}| = |\mathcal{F}_{FM}^{ID}| \wedge |FM.C| = |C_{FM}^{ID}|$
- Valid parents:  $\forall F \in FM.\mathcal{F} : F.parentID \in \mathcal{F}_{FM}^{ID} \cup \{\perp, \dagger\}$
- Valid constraints:  $\forall C \in FM.C : Var(C.\phi) \subseteq \mathcal{F}_{FM}^{ID}$
- Single root:  $\exists ! F \in FM.\mathcal{F} : F.parentID = \perp$
- Acyclic:  $\forall F^ID \in \mathcal{F}_{FM}^{ID} : F^ID \leq_{FM} \perp \vee F^ID \leq_{FM} \dagger$

We denote the set of all legal feature models as  $\mathcal{F}M$ .

We utilize this formalization of feature models for defining our operation model in Section 3.

### 2.2 Consistency Maintenance

Real-time, remote collaborative editing systems, usually rely on *operations* to propagate changes among connected users [43]. An operation is the description of an atomic manipulation of a document with a distinct intention. It is applied to a document to transform it from an old to a new (modified) state. We adopt this operational concept and use the definitions of Sun et al. [53] to formally describe *concurrency* and *conflict*.

*Concurrency.* Multiple users can create operations at different sites at different times. However, the synchronization of these operations between sites is affected by network latency, and thus not instant. Consequently, the order of submitted operations cannot be simply tracked based on physical time. Instead, we adapt a well-known strict partial order [30, 34, 53] to determine the temporal (and thus causal) relationships of operations and define the notion of concurrency.

**Definition 4** (Causally-Preceding Relation [53]). Let  $O_a$  and  $O_b$  be two operations generated at sites  $i$  and  $j$ , respectively. Then,  $O_a \rightarrow O_b$  ( $O_a$  causally precedes  $O_b$ ) iff at least one of the following is true:

- $i = j$  and  $O_a$  is generated before  $O_b$

- $i \neq j$  and at site  $j$ ,  $O_a$  is executed before  $O_b$  is generated

•  $\exists O_x : O_a \rightarrow O_x \wedge O_x \rightarrow O_b$

where before refers to a site's local physical time. Further,  $O_a$  and  $O_b$  are said to be concurrent iff  $O_a \not\rightarrow O_b$  and  $O_b \not\rightarrow O_a$ .

*Conflict.* Several challenges hamper the maintenance of a consistent document state in collaborative, real-time editors [51, 53]. Of particular interest is the *intention violation* problem, which is concerned with conflicts. A conflict occurs if two or more concurrent operations violate each other's intentions. For example, two operations that set the name of the same feature to different values are intention-violating (i.e., in conflict), as both override the other operation's effect. In Section 5, we describe how this problem may be solved in the context of feature modeling.

### 3 OPERATION MODEL

A collaborative feature model editor must support a variety of operations to achieve a similar user experience as single-user editors. However, supporting various operations can lead to more interactions between operations, which makes consistency checking and resolving of conflicts more complex. Furthermore, it hampers reasoning about the editor's correctness. To address this issue, we use a two-layered operation architecture [54], in which we separate two kinds of operations: low-level Primitive Operations (POs) and high-level Compound Operations (COs). POs represent fine-grained edits to feature models and are suitable to use in concurrency control techniques, as they are simple and composable. A CO is an ordered sequence of POs and exposes an actual feature-modeling operation to the application.

Using this two-layer architecture, instead of one large set of operations, has several advantages: When detecting conflicts between operations, we can focus on POs and do not need to analyze any high-level COs, as they are PO sequences. Also, to extend an editor with additional operations, we only need to implement new COs, without making major changes to the conflict detection. In the following, we exemplify POs and COs.

*Primitive Operations.* Single-user feature modeling tools allow creating, removing, and modifying features and cross-tree constraints in various ways. We present two exemplary POs that serve as building blocks for such COs. For each PO, we provide formal semantics in the form of pre- and postconditions, where  $FM$  and  $FM'$  refer to the feature model before and after applying the PO, respectively. By convention, no PO shall have any other side effects than those specified in the postconditions.

**createFeaturePO( $F^{ID}$ ):** Creates a feature with a globally unique identifier and default attributes, not yet inserted to the feature tree.

PRE:  $F^{ID} \in \mathcal{ID}$   
 $F^{ID} \notin \mathcal{F}_{FM}^{ID}$

POST:  $(F^{ID}, \dagger, true, and, false, NewFeature) \in FM'.\mathcal{F}$

**updateFeaturePO( $F^{ID}$ ,  $attr$ ,  $oldVal$ ,  $newVal$ ):** Updates an attribute  $attr$  of a feature  $F^{ID}$  to a new value  $newVal$ . The old attribute value is included as well to facilitate conflict detection.  $Dom$  refers to an attribute's domain (cf. Definition 1). Further,  $FM.\mathcal{F}(F^{ID}).[attr]$  refers to a particular feature attribute value as specified by  $attr$ .

PRE:  $F^{ID} \in \mathcal{F}_{FM}^{ID}$   
 $attr \in \{parentID, optional, groupType, abstract, name\}$   
 $oldVal = FM.\mathcal{F}(F^{ID}).[attr]$   
 $newVal \in Dom(attr)$

POST:  $FM'.\mathcal{F}(F^{ID}).[attr] = newVal$

For cross-tree constraints, we define analogous POs.

*Compound Operations.* To allow for high-level modeling operations, we employ COs. Each CO consists of a sequence of atomically applied POs. Further, each CO has associated preconditions and an algorithm that generates the CO's PO sequence, which must fulfill the preconditions of each comprised PO. Whenever a user requests to execute a CO, we have to check the preconditions against the current feature model  $FM$ , and then invoke the CO's algorithm with  $FM$  and any required arguments (e.g., a feature parent  $FP$ ). We then apply the generated CO locally and propagate it to other collaborators. For the sake of brevity, we only show one exemplary CO, which creates a feature below another feature:

```
function CREATEFEATUREBELOW(FM,  $F^{ID}$ ,  $FP^{ID}$ )
  Require:  $F^{ID} \in \mathcal{ID}$ ,  $F^{ID} \notin \mathcal{F}_{FM}^{ID}$ ,  $FP^{ID} \in \mathcal{F}_{FM}^{ID}$ 
  return [createFeaturePO( $F^{ID}$ ),
         updateFeaturePO( $F^{ID}$ , parentID,  $\dagger$ ,  $FP^{ID}$ )]
end function
```

We defined additional operations, such as (re)moving features and cross-tree constraints [29], and more can be designed in the future.

*Operation Application.* As POs and COs are only descriptions of manipulations on a feature model, we further need to define how to apply them to produce a new (modified) feature model.

**Definition 5.** Let  $FM \in \mathcal{FM}$ . Further, let  $PO$  and  $CO$  be a primitive and a compound operation whose preconditions are satisfied with regard to  $FM$ . Then,  $FM' = applyPO(FM, PO)$  denotes the feature model  $FM'$  that results from applying  $PO$  to  $FM$ . Further, we define  $applyCO(FM, CO)$  as the subsequent application of all primitive operations contained in  $CO$  to  $FM$  with  $applyPO$ .

We assume the tool to provide  $applyPO$ , which ensures all postconditions of primitive operations. Note that  $applyCO$  does treat every compound operation equally, which facilitates conflict detection and future extensions. Further, we can already derive that  $applyCO$  always preserves the legality of feature models (cf. Definition 3) in a single-user scenario [29].

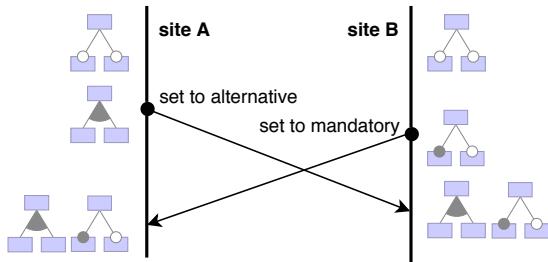
## 4 REQUIREMENTS ANALYSIS

Before developing a technique for collaborative, real-time feature modeling, we need to define the requirements that such a technique must fulfill according to the general conditions of our considered use cases. We then discuss a concurrency control technique for collaborative editing which fits our requirements.

### 4.1 Requirements

To allow users to work on the same feature model simultaneously, we define four requirements (Req) based on the general conditions we described in Section 1. This list is not complete, but focuses on formal requirements that enable our technique and its integration.

*Req<sub>1</sub>: Concurrency.* The most important requirement for enabling collaborative, real-time feature modeling is that our technique must



**Figure 1: The MVMD technique applied to feature modeling.**

allow multiple users to concurrently access and edit a model [22, 23, 26]. Consequently, our technique must incorporate a concurrency control technique to manage concurrent operations. Without such a technique, concurrency can lead to inconsistency and confusion.

*Req<sub>2</sub>: Intention Preservation.* A crucial requirement for modeling and specification activities is that an editor accurately reflects an operation’s expected behavior [13, 26, 51, 53]. This means that collaborators submit an operation and expect the system to apply and retain the intended change. To this end, our technique must ensure that the result is consistent in itself, but also to the issued operation with respect to multiple, potentially conflicting operations that are issued by various collaborators. For our technique, we require a method that prevents unexpected results, such as masked, overridden, and inconsistent operations.

*Req<sub>3</sub>: Optimism.* A collaborative, real-time editor may process operations using a *pessimistic* or *optimistic* strategy [26, 43, 53]. Pessimistic strategies require all sites to acknowledge a change before it is carried out. Thus, such strategies include additional transmissions, which block the local user interface for that time. In contrast, optimistic strategies immediately apply changes locally, then propagate them to all other sites [9, 22, 26, 43]; relying on the users to resolve all occurring conflicts afterwards—assuming that conflicts occur only sparsely [13, 20, 23, 49]. With an optimistic strategy, the local system is always responsive and allows unconstrained collaboration as long as no conflict emerges. As we aim for a small team of remotely connected collaborators, we assume that during the editing process there is noticeable network latency, but only few conflicts. Thus, an optimistic strategy seems more suited for our technique, as it most likely improves editing experience compared to pessimistic strategies.

## 4.2 Multi-Version Multi-Display Technique

In the context of feature modeling, we argue that collaborators should be involved in resolving conflicts (similar to merging in version control systems) to preserve the intentions of all conflicting operations. To this end, we use a *multi-versioning concurrency control technique* [14, 49, 55]. In contrast to other techniques, multi-versioning techniques keep different versions of objects on which conflicting operations have been performed (similar to parent commits that are merged in version control systems). In particular, we focus on the Multi-Version Multi-Display (MVMD) technique [15, 50, 51], which lets users decide which new document version should

be used in case of a conflict. In Figure 1, we show how this technique allocates two conflicting update operations on a feature model to two different versions, preserving intentions and allowing for subsequent manual conflict resolution. This technique encourages communication between collaborators and improves the confidence in the correctness of the resulting document. As MVMD fulfills all of our requirements, we use it as basis for our collaborative, real-time feature modeling technique and adapt it where necessary.

At the center of this technique is an application-specific *conflict relation*, which is used to determine algorithmically whether two operations are in conflict.

**Definition 6** (Conflict Relation [51, 56]). *A conflict relation  $\otimes$  is a binary, irreflexive, and symmetric relation that indicates whether two given operations are in conflict. If two operations  $O_a$  and  $O_b$  are not in conflict with each other, i.e.,  $O_a \not\otimes O_b$ , they are compatible. Only concurrent operations may conflict, that is, for any operations  $O_a$  and  $O_b$ ,  $O_a \parallel O_b \Rightarrow O_a \not\otimes O_b$ .*

Utilizing such a conflict relation, MVMD groups operations in suitable versions according to whether they are conflicting or compatible, as we show in Figure 1. We omit the details of how those versions may be constructed algorithmically and refer to the original paper [51]. In the following, we focus on our adaptations for feature modeling, which includes introducing a conflict relation (Section 5) as well as a conflict resolution process (Section 6) suitable for collaborative, real-time feature modeling.

## 5 CONFLICT DETECTION

In this section, we describe how we extended the MVMD technique with conflict relations for feature modeling to allow for the detection of conflicting operations. To this end, we briefly motivate and describe several required strategies and mechanisms.

### 5.1 Causal Directed Acyclic Graph

In Section 2.2, we introduced a causal ordering for tracking operations’ concurrency relationships in the system. However, the *outer* and *inner conflict relations* for collaborative feature modeling (introduced below) require further information about causality relationships. To this end, we utilize that the causally-preceding relation is a strict partial order, and thus corresponds to a directed acyclic graph [34, 47]. Using such a Causal Directed Acyclic Graph (CDAG), we define the sets of Causally Preceding (CP) and Causally Immediately Preceding (CIP) operations for a given operation as follows:

**Definition 7.** Let  $GO$  be a group of operations. The causal directed acyclic graph for  $GO$  is the graph  $G = (V, E)$  where  $V = GO$  is the set of vertices and  $E = \{(O_a, O_b) \mid O_a, O_b \in GO \wedge O_a \rightarrow O_b\}$  is the set of edges. Then, the set of causally preceding operations for an  $O \in GO$  is defined as  $CP_G(O) := \{O_a \mid (O_a, O) \in E\}$ .

Now, let  $(V, E')$  be the transitive reduction of  $(V, E)$ . Then, the set of causally immediately preceding operations for an  $O \in GO$  is defined as  $CIP_G(O) := \{O_a \mid (O_a, O) \in E'\}$ .

The transitive reduction of a graph removes all edges that only represent transitive dependencies [2]. Therefore, an operation  $O_a$  causally *immediately precedes* another operation  $O_b$  when there is no operation  $O_x$  such that  $O_a \rightarrow O_x \rightarrow O_b$ . Each collaborating site

has a copy of the current CDAG, which is incrementally constructed and includes all previously generated and received operations.

## 5.2 Outer Conflict Relation

In its original context, the MVMD technique solely uses the operations' metadata to determine conflicts. However, no complex syntactic or semantic conflicts can be detected this way, because the underlying document is not available for conflict detection. In contrast, we propose that a conflict relation for feature modeling should not only consider an operation's metadata, but also the feature model. Such a conflict relation may inspect the involved operations and apply them to a suitable feature model to check whether their application introduces any inconsistencies.

In order to guarantee that such a suitable feature model exists for two given operations, all of their causally preceding operations must be compatible. Otherwise, the intention preservation property may be violated, so that the conflict relation would rely on potentially inconsistent and unexpected feature models.

The *outer conflict relation* (termed  $\otimes_O$ ) serves to guarantee this property. It may be computed with OUTERCONFLICTING, a recursive algorithm that uses the CDAG to propagate detected conflicts to all causally succeeding operations:

```
function OUTERCONFLICTING( $G, CO_a, CO_b$ )
Require:  $G$  is the CDAG for a group of operations  $GO$ ,
 $CO_a, CO_b \in GO$ 
  if  $CO_a \nparallel CO_b \vee CO_a = CO_b$  then return false
  if  $\exists CIPO_a \in CIP_G(CO_a), CIPO_b \in CIP_G(CO_b)$ :
    OUTERCONFLICTING( $G, CIPO_a, CIPO_b$ )
     $\vee \exists CIPO_b \in CIP_G(CO_b)$ : OUTERCONFLICTING( $G, CO_a, CIPO_b$ )
     $\vee \exists CIPO_a \in CIP_G(CO_a)$ : OUTERCONFLICTING( $G, CIPO_a, CO_b$ )
  then return true
  return  $CO_a \otimes_I CO_b$ 
end function
```

In the basic case, OUTERCONFLICTING defers the conflict detection to the inner conflict relation ( $\otimes_I$ ). The other cases ensure that there is a well-defined feature model for subsequent conflict detection, which enables us to check arbitrary consistency properties; with the disadvantage that few operations may falsely be flagged as conflicting. Using OUTERCONFLICTING, we can compute  $\otimes_O$  as follows:

**Definition 8.** Two compound operations  $CO_a$  and  $CO_b$  are in outer conflict, i.e.,  $CO_a \otimes_O CO_b$ , iff  $\text{OUTERCONFLICTING}(G, CO_a, CO_b) = \text{true}$ , where  $G$  is the current CDAG at the site that executes OUTERCONFLICTING.

## 5.3 Topological Sorting Strategy

The outer conflict relation  $\otimes_O$  ensures the existence of a suitable feature model. To actually produce such a feature model, we use

$\text{applyCOs}(G, FM, COs) := \text{reduce}(\text{applyCO}, FM, \text{topsort}(COs, G))$   
 where  $\text{topsort}$  corresponds to a topological sorting of operations according to their causality relationships specified in  $G$ , which  $\text{reduce}$  then applies one by one in that order to  $FM$ . We use  $\text{APPLYCOs}$  to apply (unordered) sets of mutually compatible operations to a feature model. Because the application order of operations is important for producing a correct result, our topological sorting strategy ensures that all causal relationships captured in the CDAG are respected.

## 5.4 Inner Conflict Relation

The *inner conflict relation*  $\otimes_I$  detects conflicts that are specific to feature modeling. We introduce INNERCONFLICTING to determine  $\otimes_I$  for two given compound operations:

```
function INNERCONFLICTING( $G, FM, CO_a, CO_b$ )
Require:  $G$  is the CDAG for a group of operations  $GO$ ,
 $FM$  is the initial feature model for  $G, CO_a, CO_b \in GO$ 
  if  $CO_a \nparallel CO_b \vee CO_a = CO_b$  then return false
  if SYNTACTICALLYCONFLICTING( $G, FM, CO_a, CO_b$ )
     $\vee$  SYNTACTICALLYCONFLICTING( $G, FM, CO_b, CO_a$ )
  then return true
   $FM \leftarrow \text{APPLYCOs}(G, FM, CP_G(CO_a) \cup CP_G(CO_b) \cup \{CO_a, CO_b\})$ 
  return  $\exists SP \in \mathcal{SP} : SP(FM) = \text{true}$ 
end function
```

This algorithm makes use of SYNTACTICALLYCONFLICTING, which determines whether two COs have a *syntactic conflict* that concerns basic syntactic properties of feature models. SYNTACTICALLYCONFLICTING does so by applying both operations to a suitable feature model derived with  $\text{APPLYCOs}$  from the initial feature model. The second operation is applied step-wise, so we can inspect the current feature model for potential consistency problems using a set of *conflict detection rules* specific to feature modeling. These rules preserve the legality of feature models (cf. Definition 3) by detecting several problems, such as cycle-introducing operations and more [29].

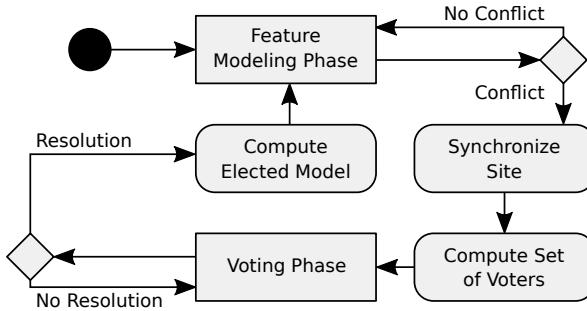
To ensure the symmetry of  $\otimes_I$ , as required by Definition 6, INNERCONFLICTING uses SYNTACTICALLYCONFLICTING to check for syntactic conflicts in both directions. Finally, INNERCONFLICTING may check additional arbitrary *semantic properties* on a feature model that includes the effects of  $CO_a$  and  $CO_b$ . A semantic property  $SP \in \mathcal{SP}$  is a deterministic function  $SP: FM \rightarrow \{\text{true}, \text{false}\}$  that returns whether a given legal feature model includes a semantic inconsistency. For instance, collaborators may want to ensure that the modeled SPL always has at least one product and does not include *dead, false-optimal features* or any *redundant cross-tree constraints* [3, 5]. Note that the MVMD technique allows only pairwise conflict detection of operations, as interactions of higher order are hard to detect [4, 11, 12]. Using INNERCONFLICTING, we can compute  $\otimes_I$  as follows:

**Definition 9.** Two compound operations  $CO_a$  and  $CO_b$  are in inner conflict, i.e.,  $CO_a \otimes_I CO_b$ , iff  $\text{INNERCONFLICTING}(G, FM, CO_a, CO_b) = \text{true}$ , where  $G$  and  $FM$  are the current CDAG and initial feature model at the site that executes INNERCONFLICTING.

Our conflict detection technique can now be implemented by using  $\otimes_O$  as conflict relation in the MVMD technique [29, 51].

## 6 CONFLICT RESOLUTION

Our extension of the MVMD technique fully automates the detection of conflicts and allocation of feature-model versions. However, MVMD does not offer functionality for actually resolving conflicts. Thus, we propose a *manual conflict resolution process* (cf. Figure 2) during which collaborators examine alternative feature model versions and negotiate a specific version [49, 55]. To this end, we allow collaborators to cast *votes* for their preferred feature model versions, which allows for fair and flexible conflict resolution [21, 25, 37].



**Figure 2: Conflict resolution process.**

In our process, a site forbids any further editing (i.e., freeze site) when a conflict is detected. This forces collaborators to address the conflict, avoiding any further divergence. The freeze also ensures the correctness of our technique, as the MVMD technique has only been proven correct for this use case [51, 56]. After freezing, the system synchronizes all sites so that all collaborators are aware of all versions before starting the voting process, which is the only synchronization period our technique needs. Next, each site may flexibly compute a set of voters (i.e., collaborators that are eligible to vote) based on the collaborators' preferences. For example, a subset could contain only collaborators involved in the conflict or those with elevated rights. To start the voting, we initialize a set of vote results as an empty set. In the voting phase, every voter may cast a vote on a single feature model version, which is added to the local vote result set and propagated to all other sites. Once cast, a vote is final and cannot be taken back, thus the vote results are a grow-only set that does not require any synchronization [48]. After a vote is processed at a site, a resolution criterion decides whether the voting phase is complete. For instance, such a criterion may involve plurality, majority or a consensus among all collaborators. When the voting phase is complete and there is a resolution, we compute the elected version from the vote result set and unfreeze the site, concluding the conflict resolution process. Otherwise, if voting is complete, but no resolution was achieved, the voting phase is restarted.

## 7 PRELIMINARY RESULTS

Although we have yet to evaluate our technique, we can already report preliminary results. Regarding the formal correctness of our technique, we can show that our technique complies with the CCI model, which is an established consistency model in literature [52, 53]. By reasoning about formal correctness, we are confident that our system allows highly-responsive, unconstrained collaboration, while still ensuring basic consistency properties.

Further, as a proof-of-concept, we have implemented our technique for collaborative, real-time feature modeling in the open-source prototype *variED*.<sup>2</sup> This web-based feature-modeling tool allows for real-time collaboration in a web browser and may serve as a basis for future user studies.

<sup>2</sup>Sources, demonstration, and information: <https://github.com/ekuiter/variED>

## 8 RELATED WORK

Closely related to our work is the *CoFM* environment that has been proposed by Yi et al. [57, 58]. With CoFM, stakeholders can construct a shared feature model and evaluate each other's work by selecting or denying model elements, resulting in a personal view for every collaborator. Our technique differs in that we only consider a single feature model, which is synchronized among all collaborators. Furthermore, we describe how to detect and resolve conflicting operations, which are not considered by CoFM. In addition, we employ optimistic replication to hide network latency, whereas CoFM uses a pessimistic approach.

Other works on feature-model editing have mostly focused on the single-user case [1, 8, 28, 35, 36]. To the best of our knowledge, none of these tools or techniques supports real-time collaboration. Rather, they allow asynchronous collaboration with version or variation control systems.

Linsbauer et al. [31] classify variation control systems, in which they notice a general lack of collaboration support compared to regular version control systems. In particular, Schwägerl and Westfechtel [45, 46] propose *SuperMod*, a variation control system for filtered editing in model-driven SPLs that supports asynchronous multi-user collaboration. However, SuperMod does not allow real-time editing and does not address conflicts that arise from the interaction of multiple collaborators.

Botterweck et al. [10] introduce *EvoFM*, a technique for modeling variability over time. Their catalog of evolution operators resembles the COs we presented in Section 3, but they do not explicitly address collaboration. Similarly, Nieke et al. [39, 40] encode the evolution of an SPL in a temporal feature model to guarantee valid configurations. With their technique, inconsistencies and evolution paradoxes can be detected. However, they do not address collaboration and provide no particular conflict resolution strategy. Change impact analyses on feature models have been proposed to identify and evaluate conflict potential of modeling decisions [17, 27, 32, 41]. These techniques do not explicitly address collaboration, but may guide collaborators in understanding and resolving conflicts.

## 9 CONCLUSION

In this paper, we presented a technique for collaborative, real-time feature modeling. Based on the general conditions of our considered use case scenarios, we defined requirements that such a technique should fulfill. Further, we described a technique for collaborative, real-time feature modeling that relies on operation-based editing and introduced primitive and compound operations. We extended the MVMD technique by introducing suitable conflict relations and a conflict resolution strategy that are suitable for feature modeling. In addition, we reported some preliminary experiences, showing the feasibility of our technique by implementing a prototype.

In future work, we want to conduct user studies to evaluate our technique. We also aim to address the question how to raise awareness of collaborators for potentially-conflicting editing operations in order to avoid conflicts in the first place.

## ACKNOWLEDGMENTS

The work of Elias Kuiter, Sebastian Krieter, and Jacob Krüger has been supported by the pure-systems Go SPLC 2019 Challenge.

## REFERENCES

- [1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming* 78, 6 (2013), 657–681.
- [2] A. V. Aho, M. R. Garey, and J. D. Ullman. 1972. The Transitive Reduction of a Directed Graph. *SIAM J. Comput.* 1, 2 (1972), 131–137.
- [3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.
- [4] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Grösslinger, and Dirk Beyer. 2013. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proceedings of the International Conference on Software Engineering*. IEEE, 482–491.
- [5] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–636.
- [6] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. 2014. Three Cases Of Feature-based Variability Modeling In Industry. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*. Springer, 302–319.
- [7] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 7:1–7:8.
- [8] Danilo Beuche. 2008. Modeling and Building Software Product Lines with Pure::Variants. In *Proceedings of the International Software Product Line Conference*. IEEE, 358–358.
- [9] Sumeer Bhola, Guruduth Banavar, and Mustaque Ahamad. 1998. Responsiveness and Consistency Tradeoffs in Interactive Groupware. In *Proceedings of the Conference on Computer-Supported Cooperative Work*. ACM, 79–88.
- [10] Goetz Botterweck, Andreas Pleuss, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. 2010. EvoFM: Feature-Driven Planning of Product-Line Evolution. In *Proceedings of the International Workshop on Product Line Approaches in Software Engineering*. ACM, 24–31.
- [11] T. F. Bowen, F. S. Dvorack, C. H. Chow, N. Griffeth, G. E. Herman, and Y.-J. Lin. 1989. The Feature Interaction Problem in Telecommunications Systems. In *Proceedings of the International Conference on Software Engineering for Telecommunication Switching Systems*. IET, 59–62.
- [12] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks* 41, 1 (2003), 115–141.
- [13] Jeffrey Dennis Campbell. 2000. *Consistency Maintenance for Real-Time Collaborative Diagram Development*. Ph.D. Dissertation. University of Pittsburgh.
- [14] David Chen. 2001. *Consistency Maintenance in Collaborative Graphics Editing Systems*. Ph.D. Dissertation. Griffith University.
- [15] David Chen and Chengzheng Sun. 2001. Optional Instant Locking in Distributed Collaborative Graphics Editing Systems. In *Proceedings of the International Conference on Parallel and Distributed Systems*. IEEE, 109–116.
- [16] Lianping Chen and Muhammad Ali Babar. 2011. A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *Information and Software Technology* 53, 4 (2011), 344–362.
- [17] Hyun Cho, Jeff Gray, Yuanfang Cai, Sonny Wong, and Tao Xie. 2011. Model-Driven Impact Analysis of Software Product Lines. In *Model-Driven Domain Analysis and Software Development: Architectures and Functions*, Janis Osis and Erika Asrina (Ed.). IGI Global, Chapter 13, 275–303.
- [18] Krzysztof Czarnecki. 2013. Variability in Software: State of the Art and Future Directions. In *Fundamental Approaches to Software Engineering (FASE)*. Springer, 1–5.
- [19] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 173–182.
- [20] Gabriele D'Angelo, Angelo Di Iorio, and Stefano Zacchiroli. 2018. Spacetime Characterization of Real-Time Collaborative Editing. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 41:1–41:19.
- [21] Alan R. Dennis, Sridhar K. Pootheri, and Vijaya L. Natarajan. 1998. Lessons from the Early Adopters of Web Groupware. *Journal of Management Information Systems* 14, 4 (1998), 65–86.
- [22] Prasun Dewan, Rajiv Choudhary, and Honghai Shen. 1994. An Editing-Based Characterization of the Design Space of Collaborative Applications. *Journal of Organizational Computing* 4, 3 (1994), 219–239.
- [23] Clarence Ellis, Simon Gibbs, and Gail Rein. 1991. Groupware: Some Issues and Experiences. *Commun. ACM* 34, 1 (1991), 39–58.
- [24] Thomas Fogdal, Helene Scherrebeck, Juha Kuusela, Martin Becker, and Bo Zhang. 2016. Ten Years of Product Line Engineering at Danfoss: Lessons Learned and Way Ahead. In *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, 252–261.
- [25] Simon Gibbs. 1989. LIZA: An Extensible Groupware Toolkit. In *Proceedings of the Conference on Human Factors in Computing Systems*. ACM, 29–35.
- [26] Saul Greenberg and David Marwood. 1994. Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface. In *Proceedings of the Conference on Computer-Supported Cooperative Work*. ACM, 207–217.
- [27] Ines Hajri, Arda Goknil, Lionel C. Briand, and Thierry Stephany. 2018. Change Impact Analysis for Evolving Configuration Decisions in Product Line Use Case Models. *Journal of Systems and Software* 139 (2018), 211–237.
- [28] Charles W. Krueger. 2007. BigLever Software Gears and the 3-Tiered SPL Methodology. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM, 844–845.
- [29] Elias Kuiter. 2019. *Consistency Maintenance for Collaborative Real-Time Feature Modeling*. Bachelor Thesis. University of Magdeburg.
- [30] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.
- [31] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *Proceedings of the International Conference on Generative Programming and Component Engineering*. ACM, 49–62.
- [32] Jihen Maâzoun, Nadia Bouassida, and Hanène Ben-Abdallah. 2016. Change Impact Analysis for Software Product Lines. *Journal of King Saud University – Computer and Information Sciences* 28, 4 (2016), 364–380.
- [33] Christian Manz, Michael Stupperich, and Manfred Reichert. 2013. Towards Integrated Variant Management In Global Software Engineering: An Experience Report. In *International Conference on Global Software Engineering*. IEEE, 168–172.
- [34] Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. North Holland, 215–226.
- [35] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.
- [36] Marcilio Mendonça, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM, 761–762.
- [37] Meredith Ringel Morris, Kathy Ryall, Chia Shen, Clifton Forlines, and Frederic Vernier. 2004. Beyond "Social Protocols": Multi-user Coordination Policies for Co-Located Groupware. In *Proceedings of the Conference on Computer-Supported Cooperative Work*. ACM, 262–265.
- [38] Damir Nešić, Jacob Krüger, Stefan Stănculescu, and Thorsten Berger. 2019. Principles of Feature Modeling. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM.
- [39] Michael Nieke, Christoph Seidl, and Sven Schuster. 2016. Guaranteeing Configuration Validity in Evolving Software Product Lines. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 73–80.
- [40] Michael Nieke, Christoph Seidl, and Thomas Thüm. 2018. Back to the Future: Avoiding Paradoxes in Feature-Model Evolution. In *Proceedings of the International Software Product Line Conference*. ACM, 48–51.
- [41] Paulius Paskevicius, Robertas Damasevicius, and Vytautas Štuikys. 2012. Change Impact Analysis of Feature Models. In *Proceedings of the International Conference on Information and Software Technologies*. Springer, 108–122.
- [42] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.
- [43] Atul Prakash. 1999. Group Editors. In *Computer Supported Co-Operative Work*, Michel Beaudouin-Lafon (Ed.). Wiley, Chapter 5, 103–134.
- [44] Rüdiger Schollmeier. 2001. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. In *Proceedings of the International Conference on Peer-to-Peer Computing*. IEEE, 101–102.
- [45] Felix Schwägerl and Bernhard Westfechtel. 2016. Collaborative and Distributed Management of Versioned Model-Driven Software Product Lines. In *International Joint Conference on Software Technologies*. SciTePress, 83–94.
- [46] Felix Schwägerl and Bernhard Westfechtel. 2017. Maintaining Workspace Consistency in Filtered Editing of Dynamically Evolving Model-Driven Software Product Lines. In *Proceedings of the International Conference on Model-Driven Engineering and Software Development*. SciTePress, 15–28.
- [47] Reinhard Schwarz and Friedemann Mattern. 1994. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing* 7, 3 (1994), 149–174.
- [48] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A Comprehensive Study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt.
- [49] Mark Steffil, Gregg Foster, Daniel G. Bobrow, Kenneth Kahn, Stan Lanning, and Lucy Suchman. 1987. Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings. *Commun. ACM* 30, 1 (1987), 32–47.
- [50] Chengzheng Sun and David Chen. 2000. A Multi-Version Approach to Conflict Resolution in Distributed Groupware Systems. In *Proceedings of the International Conference on Distributed Computing Systems*. IEEE, 316–325.
- [51] Chengzheng Sun and David Chen. 2002. Consistency Maintenance in Real-Time Collaborative Graphics Editing Systems. *ACM Transactions on Computer-Human Interaction* 9, 1 (2002), 1–41.
- [52] Chengzheng Sun and Clarence Ellis. 1998. Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. In *Proceedings of the*

- Conference on Computer-Supported Cooperative Work*. ACM, 59–68.
- [53] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. 1998. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction* 5, 1 (1998), 63–108.
  - [54] Chengzheng Sun, Steven Xia, David Sun, David Chen, Haifeng Shen, and Wentong Cai. 2006. Transparent Adaptation of Single-User Applications for Multi-User Real-Time Collaboration. *ACM Transactions on Computer-Human Interaction* 13, 4 (2006), 531–582.
  - [55] Volker Wulf. 1995. Negotiability: A Metaprogram to Tailor Access to Data in Groupware. *Behaviour & Information Technology* 14, 3 (1995), 143–151.
  - [56] Liyin Xue, Mehmet Orgun, and Kang Zhang. 2003. A Multi-Versioning Algorithm for Intention Preservation in Distributed Real-Time Group Editors. In *Proceedings of the Australasian Computer Science Conference*. ACS, 19–28.
  - [57] Li Yi, Wei Zhang, Haiyan Zhao, Zhi Jin, and Hong Mei. 2010. CoFM: A Web-Based Collaborative Feature Modeling System for Internetwork Requirements’ Gathering and Continual Evolution. In *Proceedings of the Asia-Pacific Symposium on Internetwork*. ACM, 23:1–23:4.
  - [58] Li Yi, Haiyan Zhao, Wei Zhang, and Zhi Jin. 2012. CoFM: An Environment for Collaborative Feature Modeling. In *Proceedings of the International Requirements Engineering Conference*. IEEE, 317–318.

# Process Mining to Unleash Variability Management: Discovering Configuration Workflows Using Logs

Ángel Jesús Varela-Vaca, José A. Galindo, Belén Ramos-Gutiérrez

María Teresa Gómez-López and David Benavides

Universidad de Sevilla

Seville, Spain

{ajvarela,jagalindo,brgutierrez,maytegomez,benavides}@us.es

## ABSTRACT

Variability models are used to build configurators. Configurators are programs that guide users through the configuration process to reach a desired configuration that fulfills user requirements. The same variability model can be used to design different configurators employing different techniques. One of the elements that can change in a configurator is the configuration workflow, i.e., the order and sequence in which the different configuration elements are presented to the configuration stakeholders. When developing a configurator, a challenge is to decide the configuration workflow that better suits stakeholders according to previous configurations. For example, when configuring a Linux distribution, the configuration process starts by choosing the network or the graphic card, and then other packages with respect to a given sequence. In this paper, we present COfiguration workfLOw proceSS mIning (COLOSSI), an automated technique that given a set of logs of previous configurations and a variability model can automatically assist to determine the configuration workflow that better fits the configuration logs generated by user activities. The technique is based on process discovery, commonly used in the process mining area, with an adaptation to configuration contexts. Our proposal is validated using existing data from an ERP configuration environment showing its feasibility. Furthermore, we open the door to new applications of process mining techniques in different areas of software product line engineering.

## CCS CONCEPTS

- Software and its engineering → Software product lines.

## KEYWORDS

variability, configuration workflow, process mining, process discovery, clustering

### ACM Reference Format:

Ángel Jesús Varela-Vaca, José A. Galindo, Belén Ramos-Gutiérrez and María Teresa Gómez-López and David Benavides. 2019. Process Mining to Unleash Variability Management: Discovering Configuration Workflows Using Logs. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3336294.3336303>

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336303>

## 1 INTRODUCTION

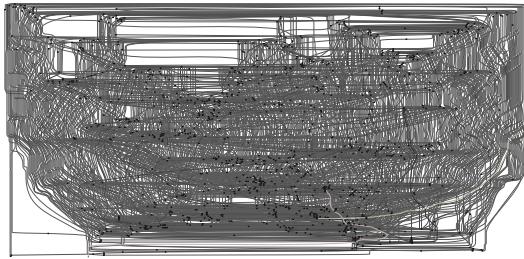
Variability models such as Feature Models (FMs) [22] describe commonalities and variabilities in Software Product Lines (SPLs) and are used along all the SPL development process. After an FM is defined, products can be configured and derived. In the configuration and derivation process, users select and deselect features using a *configurator*. A configurator [21][19] is a software tool that presents configuration options to the users in different stages. An example of a configurator tool is KConfig [58] where developers can configure the Linux kernel with more than 12.000 configuration options.

An important aspect of a configurator is to determine the *configuration workflow* [28], i.e., the order in which features and options are presented to configuration stakeholders. For instance, when configuring the Linux kernel using KConfig [58], there can be different user configuration profiles depending on interests or skills. The configuration workflow used by a configurator can impact the user experience in the configuration process. Therefore, selecting a well suited configuration workflow is a challenge. Up to now – to the best of our knowledge – the selection of a configuration workflow is made either intuitively or following the structure and properties of a variability model [21, 66].

In this paper, we present COLOSSI, an approach that takes a feature model and a set of existing configuration logs and automatically retrieves configuration workflows. A configuration log is a set of configurations performed in the past in a given domain taking into account a configuration order. Our solution relies on *process mining* [3] techniques. Process mining is a well established area of business process management that uses different techniques to extract business processes from traces of execution. In our approach, we conceptually map a business process model to a configuration workflow and traces to configuration logs making possible to reuse process mining techniques to infer configuration workflows.

Although using process mining can automatically retrieve configuration workflows, the results can be difficult to interpret to domain engineers in order to build a configurator. This is mainly due to the fact that, very often, mined processes are “spaghetti-like” models in which the same activity needs to be duplicated [62]. To illustrate the difficulty, Figure 1 shows the result of directly applying process mining techniques to the ERP system presented in [46] and detailed in Section 3.

The simplification of spaghetti processes is an open problem in the process mining domain [3]. Variability models have special characteristics that can help to guide the discovery process. To overcome this difficulty, our solution adapts a clustering algorithm



**Figure 1: Spaghetti process of the ERP presented in [46].**

solution that instead of retrieving a single-complicated configuration workflow, is able to cluster the configurations according to different metrics. Our solution takes information from the variability model as input and retrieves less complex configuration workflows that can assist the development of better configurators.

COLOSSI is validated using an ERP case study taken from [46]. Results show that the metrics of the retrieved configuration workflows are improved by a 99% in the best case and 81% in the worst case with respect to the spaghetti like first solution.

The contributions of this paper are as follows:

- An automated technique based on process mining to select a configuration workflow that better fits the configuration logs according to a set of metrics.
- A validation of the proposal using a realistic ERP scenario.
- An available implementation that can be applied to other datasets.

The remainder of this paper is organised as follows: Section 2 details the solution and concepts that grounds our proposal; Section 3 presents empirical results from analysing COLOSSI; Section 4 presents the related work and Section 5 presents concluding remarks and lessons learned.

## 2 COLOSSI: CONFIGURATION WORKFLOW PROCESS MINING SOLUTION

In order to create a configuration workflow, a feature model and a configuration log must be combined. Figure 2 shows an overview of the COLOSSI approach. Using the configuration log, it is possible to apply process mining techniques to derive a valid configuration workflow representing all the possible paths defined in the configuration logs. It is likely that the resulting workflow follows the so-called spaghetti-style [62] and it is therefore difficult to understand and manipulate. Nevertheless, it is important to remark that it can be already exploited by process mining automated tools to extract metrics, perform simplification over the workflow as well as many additional analysis. Also, any generated configuration workflow can be already used to build automatically a configurator.

In addition, to the usage of process mining techniques, we propose handling and clustering methods to reduce and group similar configuration traces according to some properties. Those clusters can then be used again as input of process mining techniques to obtaining a set of configuration workflows depending on the observed behaviour of the configuration logs. Those workflows will

obtain better metrics with respect to the original complex workflows of step ①. Our conjecture is that the resulting configuration workflows of step ② will better guide the domain engineers in the construction of a configurator as well as the analysis mentioned previously.

Following, we describe the details of the different elements of COLOSSI.

### 2.1 Inputs

A feature model is an arranged set of features that describes variability and commonality using features and relationships among them. [18, 57]. FMs describe all the potential combinations of features. Figure 3 shows an excerpt of a feature model of the ERP domain where features are arranged in a tree-like structure and different relationships are established among them. FMs can be used to build configurators that are pieces of software that guide the configuration process while selecting and deselecting features. An example of a configurator is KConfig, a tool that helps configuring the Linux kernel. As an FM can define a configuration space defined by all the possible feature combinations, it can also define different possible configuration workflows that can be derived using the same FM.

COLOSSI takes as input a FM and a *configuration log*. To define a configuration log, we use some definitions that are used in process mining area to define events and traces and we map those definitions to define a configuration log.

An event log is a multiset of traces:

*Definition 2.1.* (Event Log). Let  $L$  be an event log  $L = [\tau_1, \dots, \tau_m]$  as a multiset of traces  $\tau_i$ .

A trace is a tuple with an identifier and a sequence of events that occurred at some point in time:

*Definition 2.2.* (Trace). Let  $\tau$  be a trace  $\tau = \langle \text{case\_id}, \mathcal{E} \rangle$  which consists of a *case\_id* which identifies the case, and a sequence of events  $\mathcal{E} = \{\varepsilon_1, \dots, \varepsilon_n\}$ ,  $\varepsilon_i$  occurring at a time index  $i$  relative to the other events in  $\mathcal{E}$ .

An event occurrence is a 3-tuple with an identifier of an activity that occurred at some timestamp and that can have additional information:

*Definition 2.3.* (Event occurrence). Let  $\varepsilon$  be an event occurrence  $\varepsilon = \langle \text{activity\_id}, \text{timestamps}, \text{others} \rangle$  which is specified by the identity of an activity which produces it and the timestamps. It can store more information (i.e., states, labels, resources, etc.)

In COLOSSI, we conceptually map elements from the feature modelling domain to the process mining domain as shown in Table 1. Concretely, an event log is conceptually a configuration log. A trace is an ordered configuration, i.e., a *configuration trace*, thus, it is a set of selected features that follow a given order. Finally, an event occurrence is a feature. Additionally, a feature can have more information like attributes associated with this feature such as preferences, metrics or the like.

### 2.2 Configuration logs extractor

A configuration log is composed of a set of configuration traces where each configuration trace encodes not only the features of a configuration but the timestamps indicating when each feature

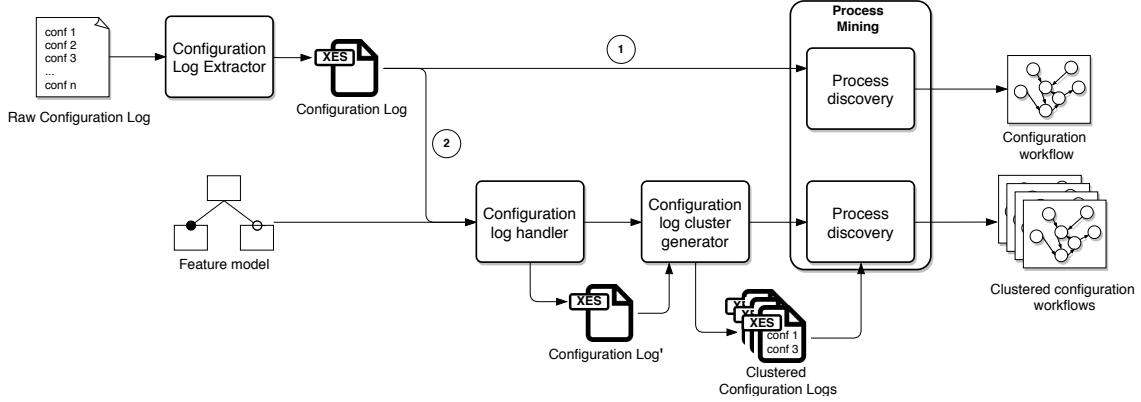


Figure 2: COLOSSI solution overview.

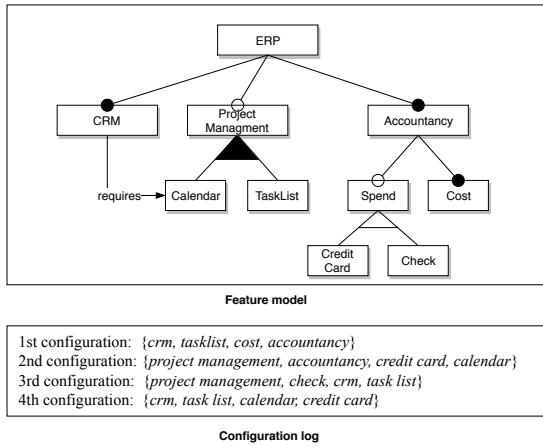


Figure 3: ERP domain based example.

Process Mining	Product Line
Event log	Configuration log
Trace	Configuration trace
Event occurrence	Feature

Table 1: Mapping concepts.

was selected. In a raw configuration log, we can find a diversity of meta-information among the selected or deselected features. Moreover, this meta-information can be presented in a unstructured or structured fashion.

In this first step, we take as input a raw configuration log and output a set of configuration traces. Therefore, we need to *i*) search for the meta-information encoding the timestamps for each feature. Note that this might not be explicit and can be provided using other mechanisms (e.g., line numbers in a plain text format); *ii*) use this meta-information to represent the feature selection order; and *iii*) store the set of configuration traces in a format that can be used throughout the configuration workflow retrieval process (e.g., XES serialization [1]). After this, we end up with a set of configuration traces that represent the selection order used by the configurator users. However, there might be non-valid configurations and other erroneous configurations w.r.t domain information.

### 2.3 Configuration logs handler

At this step, the configuration log might contain non-valid configurations, erroneous partial selection of features among other domain-related errors such as those depicted in [6]. To remove clutter and noise out of the workflows, users might prefer to remove such information from the configuration log. This cleaning step consist on removing wrong selection of features (a.k.a non-valid partial configurations) as well as generate metrics that can be latter exploited to optimise the workflow retrieval process. For example, the use of atomic-sets to complete partial configurations.

Depending on the expected workflow usage, domain engineers have to define the meaning of a valid configuration and the metrics to rely on. For example, a SPL engineer might consider only configurations with complete assignments of features to develop a configuration while other might find interesting to consider full assignments (i.e., to configure only the variability part of the product line, keeping aside the common parts).

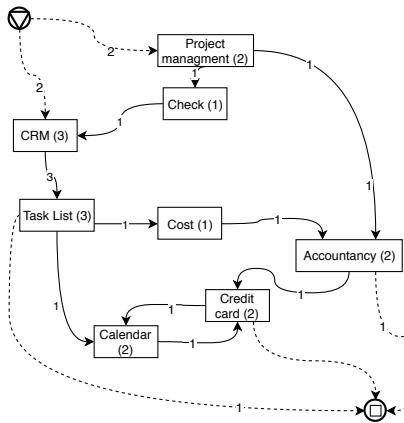
### 2.4 Process mining - discovery

Process mining is a family of techniques based on event logs that can be categorised as process discovery, conformance checking and enhancement [63]. In this paper, we are focused on the use of *process mining* to analyse the configuration logs for the discovering of configuration workflows based on the user experiences. Process discovery in process mining brings together a set of algorithms to generate a workflow process model that covers the traces of activities observed in an organisation [40]. The evolution of algorithms during last decades has allowed the discovery of complex models that are able to involve not only the activities executed in the daily work of companies, but also the persons who execute them and the used resources.

Process mining is an important topic that has been well received by the enterprises, bringing about the evolution of the research solution tools (e.g., ProM [64]) to commercial solutions (e.g., Disco™ and Celonis™). This facilitates its applicability to several contexts and areas, although variability has been out of the scope of these techniques before this paper.

Process discovery in process mining uses a set of traces similar to the configuration log shown in Figure 3, to obtain a model that

covers the possible traces. Figure 4 shows the process discovered by Disco tool-suite, which covers every possibility configuration trace. The relational patterns among the definition of the features become part of the model. For example, two features can be the first in the traces (*CRM* or *Project management*) or after *CRM* always *Task List* is selected. Figure 4 also shows the number of traces that are represented by each transition, giving information about the importance of each part of the traces in the obtained model.



**Figure 4: Process discovered for configuration log of ERP domain based example.**

In the framework proposed in this paper (Figure 2), *Process mining - process discovery* module enables to read an event log and generates a process model that fits these traces. In the case of the variability context, a *configuration log* is read and a *configuration workflow* is obtained using the same techniques used for classical process mining.

## 2.5 Configuration logs cluster generator

Configuration processes have a high degree of variability, specially when the configuration order is defined by human decisions. The application of process discovery in this type of scenarios tends to produce spaghetti-like processes, being necessary to apply a pre-processing. Configurability contexts are specially variable in relation to the executed activities derived from the high human intervention, thereby, we propose to divide the traces into subsets, to model different profiles of users and avoiding the discovery of non-user understandable processes. In these contexts, where process discovery is used to infer spaghetti-like processes, frequently clustering techniques such as a pre-processing step are applied [26]. To adapt the solution to configuration tasks, we propose the division of the configuration traces into multiple clusters before the application of a process discovery. This division lets to discover configuration workflows with more quality. This section describes what a cluster is and the metric (e.g., entropy) used to divide the traces among them. In following sections, we describe how quality is measured and how the clusters can be created.

Being  $L$  a configuration log composed of a set of configuration traces (i.e.,  $[\tau_1, \dots, \tau_m]$ ), a cluster is a subset of configuration traces from  $L$  that complies certain properties.

**Definition 2.4.** (Cluster of Configuration Traces). A cluster of configuration logs,  $c = [\tau_1, \dots, \tau_j] \subseteq L$ , where  $\forall \tau_k \subseteq L, \exists c \mid \tau_k \in c$  and  $\nexists c' \neq c$  where  $\tau_k \in c'$ .

The distribution of configuration traces between various clusters depends on the purpose of the practitioners. In our case, the goal is to group the more similar configuration traces. In this paper, the meaning of 'similar' is related to both features and transitions involved in the logs. For this reason, we adapted the classical information entropy metric [37] by introducing two different custom entropy metrics for clustering in the configuration context:

- *Entropy-features* ( $S_{features}$ ) of a cluster: a metric which measures the similarity between a set of traces according to the features that belong to the same cluster. Thus, it is the ratio between the number of features that do not appear in all configuration traces ( $features_{nat}$ ) and the number of different features in all the configuration traces ( $features_{diff}$ ):

$$S_{features} = \frac{|features_{nat}|}{|features_{diff}|} \quad (1)$$

- *Entropy-transitions* ( $S_{transitions}$ ) of a cluster: a metric which measures the similarity between a set of traces according to the transitions that belong to the same cluster. Thus, it is the ratio between the transitions that do not appear in all configuration traces ( $transitions_{nat}$ ) and the number of different transitions in all the configuration traces ( $transitions_{diff}$ ):

$$S_{transitions} = \frac{|transitions_{nat}|}{|transitions_{diff}|} \quad (2)$$

In order to illustrate the calculation of entropies, the  $S_{features}$  and  $S_{transitions}$  for the *Cluster 1* and *Cluster 2* of Figure 5 are determined in Table 2.

	Entropy-features	Entropy- transitions
Cluster 1	$\frac{0}{4} = 0$	$\frac{0}{3} = 0$
Cluster 2	$\frac{6}{8} = 0,75$	$\frac{6}{15} = 0,4$

**Table 2: Entropies for the clusters of the Figure 5.**

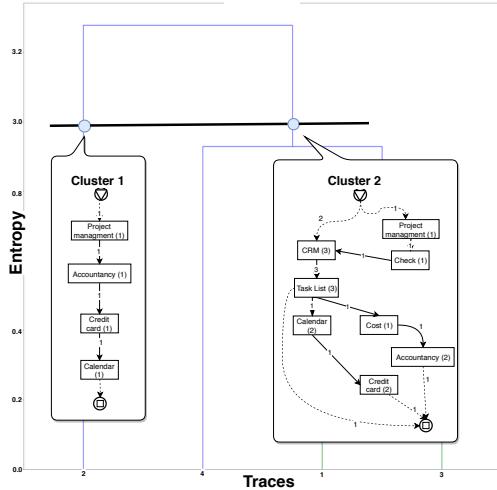
Note that the range of the entropy is  $[0..1]$ . The values of entropy that are close to 0 represent more similar traces, whilst when they are close to 1 represent that there are different features involved in the traces of the cluster. The best configuration of clusters obtained from a set of configurations traces is the one that minimize the summation of the entropy of all clusters obtained. The challenge is how to obtain the best configuration of clusters as a pre-processing of a process discovery.

In order to find out the best configuration traces divided into clusters, minimising the entropy of the resulting clusters, different algorithms for clustering can be used. In accordance with [30] clustering provides an unsupervised classification of patterns (observations, data items, or feature vectors) into groups (clusters). Clustering [31] brings together a large set of algorithms that can be classified in different ways according to the point of view necessary in the case of study. We propose the use of the well-known *hierarchical agglomerative clustering* algorithm base on the work in [69] as the combination of hierarchical and agglomerative clustering.

On the first hand, *hierarchical clustering* is defined as a procedure to form hierarchical groups of mutually exclusive subsets, each of which has members that are maximally similar with respect to the specified characteristics [69]. In the same study, authors define the process as: assuming we start from  $n$  sets, it permits their reduction to  $n - 1$  mutually exclusive sets by considering the union of all possible  $\frac{n(n-1)}{2}$  pairs and selecting a union having a maximal value for the objective function.

On the other hand, *agglomerative clustering* is an algorithm that starts from the assumption that each element constitutes a cluster by itself (singleton) and it successively merges these singletons together forming clusters until a stopping criterion is satisfied which is also determined by the objective function.

The characteristics used in our solution is based on both entropies presented (features and transitions), combined with the objective function the Ward's minimum variance method [69]. This function aims to minimise the sum of the squared differences within all clusters, which means, a variance-minimising approach. Figure 5 shows the obtained dendrogram<sup>1</sup> for the example in Figure 3 by using Entropy-features.



**Figure 5: Clustering for the ERP excerpt using the *Entropy-features*.**

It is well-known that every clustering algorithm builds a distance matrix during its execution based on a given methodology (euclidean, manhattan, etc.). However, *hierarchical agglomerative clustering* is an exception, since it can be carried out from the distance matrix itself. We consider the entropy matrix as the distance matrix, this leads us to decide for this method of clustering, which can be performed also using other methods, such as *single-linkage*, *complete-linkage*, *average-linkage*, and *Ward*.

Whenever a clustering process is executed, one of the first problems to deal with is to decide which is the optimal number of clusters. Many studies carried out about the discovery of the optimum number of clusters, therefore, we decided to study a significant

<sup>1</sup>Dendrogram is a branching diagram which represents the arrangement of the clusters produced by the corresponding analyses

number of them to choose by voting the number of clusters that most indicators had selected as optimal. In this regard, 17 different indicators [4, 5, 9, 14, 16, 17, 20, 24, 25, 29, 32, 33, 42, 44, 45, 50, 51] are used as reference to choose the best number of clusters adapted to our scenario.

By using these indicators with the different clustering methods mentioned above, all of them selected a too large number of clusters as the optimal solution. In addition, the values of the indicators themselves are in some of the cases too dissimilar.

For all the methods the range [0-10] is proposed to determine the number of clusters. In the *single-linkage* method always retrieved the maximum (i.e., 10). For the *average-linkage* retrieved always an average of 2 clusters. Both methods help to bound the limits of number of clusters between 2 and 10. Only with the *complete-linkage* and *Ward* methods, more homogeneous, similar and an assumable number of clusters as results were obtained for the dendograms showed in Figures 6 and 7.

Finally, by making use of dendograms, it is observed that applying the *Ward* method, the samples becomes better distributed among the clusters, thereby, generating more differentiated clusters and better structured dendograms. This fact determines the *Ward*'s method as the best option for our approach.

## 2.6 Leveraging the results of COLOSSI

The COLOSSI approach can be used in different scenarios to leverage process mining in variability management. One of the scenarios presented in this paper is the building of configurators. However, we envision other areas where process mining can be used to automate different tasks. Next, we describe those scenarios, also related to software product lines, from our experience and perspective:

- **Configurator building.** Up to now, configurators building is performed using manual mechanisms or, at most, using the information present in the variability model (e.g., tree traversal in feature models). With COLOSSI, we open the door to use existing configuration logs to build configurators. This novel approach can open the door to new ways of assisting configurators builders by using the generated configuration workflow to optimise configurators.
- **Data analysis.** From the generated configuration workflow it is possible to perform many analysis in terms of graph metrics. Deadlocks identifications, misalignment analysis, metrics extraction –to just mention a few– are areas where process mining techniques can be useful.
- **Testing.** From the data extracted in the former item, it could be possible to define new sampling techniques [61] that can improve the identification of bugs or feature interactions in existing product lines.
- **Variability reduction.** One of the challenges for companies that develop software product lines is variability reduction [8]. While variability is a must in a software product line approach, it is always difficult to find a trade off between a high degree of variability and a systematic management of such a variability. In this context, experts claim for techniques and tools to reduce variability while preserving configurability. Process mining techniques presented in this paper can be a

first step towards defining tools to assist in the decision of variability reduction.

- **Reverse engineering.** One of the inputs used when reverse engineering feature models are configurations (a.k.a. product matrix). We envision that the techniques described in this paper can be used in reverse engineering of variability models. For instance, the generated configuration workflow can be analysed to better guide reverse engineering algorithms

### 3 EVALUATION

In this section we present the evaluation of COLOSSI. The evaluation consists of the application of the framework detailed in Section 2 to a configuration log obtained from a real scenario. The possible clusters derived from the application of the defined entropies are analysed.

#### 3.1 Experimentation data

In order to analyse the applicability of our example in a configuration real scenario, we used the raw information from [46]. The used data include a configuration model representing a real ERP, as well as a raw configuration log. The ERP feature model has 1920 features and 59044 cross-tree constraints. Also, the configuration log is formed of 35193 event occurrences that represent a total of 170 different configuration traces with an average of 207 features per configuration trace.

#### 3.2 Framework application

In this section we detail each task of the framework presented in Figure 2.

**3.2.1 Configuration Log Extractor.** The input data of the configuration of the ERP is represented in a CSV file with two elements, the configuration id and the feature that is configured. Note that a feature can appear in one or more traces, but no more than once in the same trace. Then, the timestamp required to extract the traces was taken by the line number in which the features were appearing throughout the file in a sequential order. This is, we assume that the timestamps were implicit based on the order of appearance (i.e., line numbers) then, transformed them into a more standard format for traces. Concretely we use in our solution the IEEE Standard for eXtensible Event Stream (XES) [1]. This is a standard to serialise, store, exchange events data and it is commonly used in process mining techniques.

**3.2.2 Configuration Log Handler.** To clean up the set of configurations retrieved by the extractor we decided to consider only valid partial and full configurations. This filtering operation is performed by using the FaMa framework [7]. After filtering, the valid partial configurations using automated analysis [6], we ended up considering 61 configuration traces from the initial set of 170.

**3.2.3 Configuration Log Cluster Generator and Process Discovery.** Figure 6 and 7 represent the obtained clusters according to the dendrogram built by means of the entropy of features and transitions analysis respectively. In the case of feature entropy (Figure 6), five clusters are obtained. On the other hand, when the transition entropy is used, three clusters are derived to split the configuration

logs into simpler configuration workflows. In the following subsections, the details about the obtained configuration workflows and clusters are analysed.

#### 3.3 Analysis of Results

In order to evaluate how the application of clustering can improve the configuration workflows obtained by COLOSSI, in this section, we compare the models discovered: (1) the *original* configuration logs obtained from the initial raw configuration log (cf. Section 3.1); (2) the *filtered* version of the same log including only valid configurations (i.e., after applying *configuration log handler*), and; (3) two set of clusters based on the proposed entropies (features and transitions explained in Section 2.5).

The analysis is carried out following two different perspectives: (1) the analysis of the discovered configuration workflows and (2) the analysis of the set of configuration traces involved in each cluster used in the process discovery.

**3.3.1 Analysis of discovered configuration workflows.** First, highlight that inductive process discovery techniques used by COLOSSI, ensure the soundness and correctness of the process models obtained [35]. Thus, an analysis of the soundness and correctness of the configuration workflows are unnecessary since processes discovered is always complete, have a proper completion, and have no dead transitions.

However, the complexity of the configuration models is affected by the number of features, the number of configuration traces and the number of transitions. Obviously, the filtering of the configuration traces or the division of the logs will bring about simpler configuration workflows. Figure 8 depicts in a comparative way the number of features, configuration traces and transitions of the set of configuration logs using in each scenario: original configuration log, filtered configuration log only with valid traces, the 5 clusters obtained by using entropy-features, and the 3 clusters obtained by using entropy-transitions.

Regarding general parameters, there is an enormous difference between the original, filtered version, and both that use clustering. The original configuration workflow contains 1652 features and 3330 transitions, whilst the filtered version has one less magnitude order of features and transitions. The clusters seem similar to the filtered version, however, each cluster contains, at least, less than half regarding features. In case of the transitions, the filtered version reached four times fewer transitions than the original. However, clusters reached in the worst case 500 transitions less than the filtered and more than one third fewer transitions in the best case. Due to the clusters group a set of similar traces, the number of traces is intrinsically smaller regarding the configuration traces of the original and filtered configuration workflow.

In conclusion, clusters enable to reduce the complexity of configuration workflow discovered by reducing the configuration traces involved in the same configuration workflow. However, the question is what level the quality of the obtained workflow is improved, and which distribution of cluster-entropy works better.

In literature, several metrics are used to measure how "good" is a design of a business process model [10, 43, 48]. Discovered configuration workflows are also processes with features instead of activities, therefore, these metrics can be adapted to measure

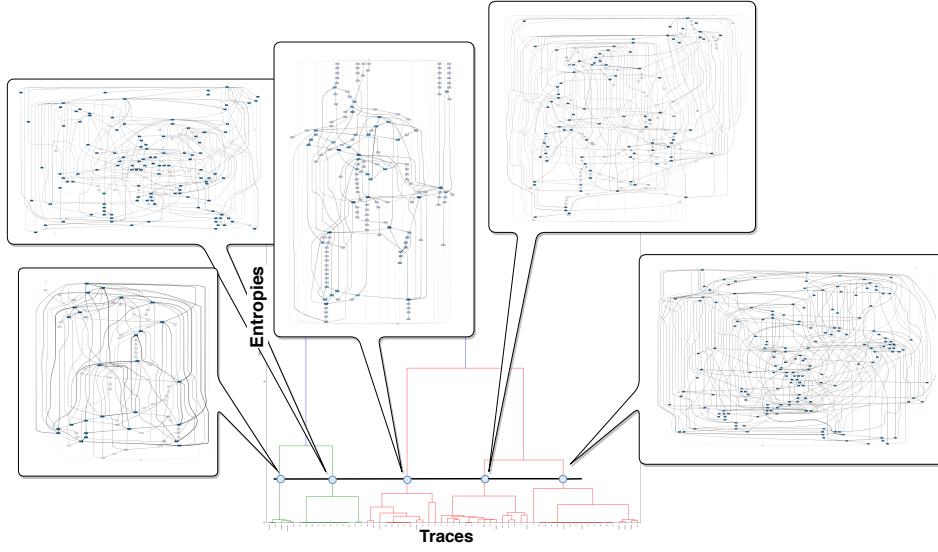


Figure 6: Clustering for the ERP example using the *Entropy-features*.

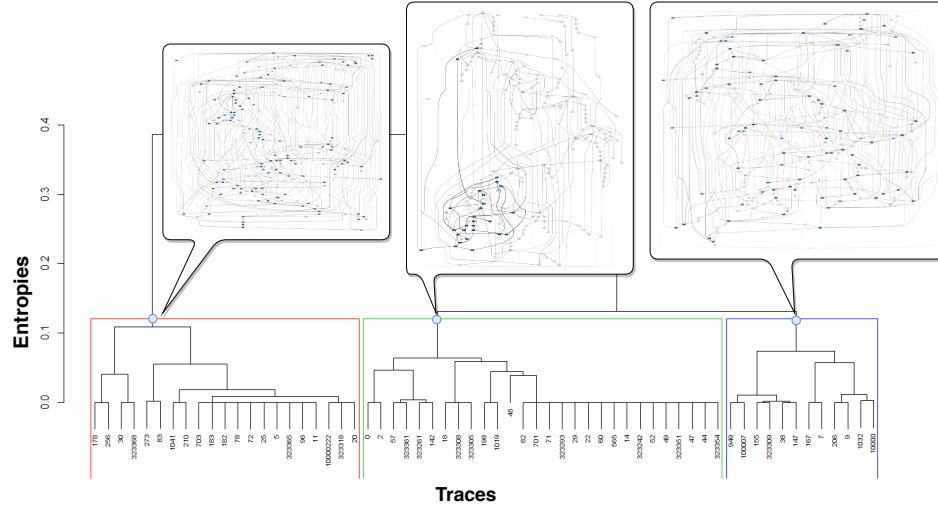


Figure 7: Clustering for the ERP example using the *Entropy-transitions*.

the quality of our obtained configuration workflows. The next set of metrics is adapted to measure the understandability and the complexity of the configuration workflows to compare the four discovered configuration workflows:

- *Density*: the ratio of transitions divided by the maximum number of possible transitions. The lower the value of density, the higher the understandability and the lower complexity.
- *Cyclomatic number (CC)*: the number of paths needed to visit all features. The cyclomatic can be seen as a complexity

metric, thus, the lower the value of *CC*, the lower the level of complexity.

- *Coefficient of connectivity (CNC)*: the ratio of transitions to features. The greater the value of *CNC*, the greater the complexity of configuration workflows. Although, the authors in [43] remark that models with the same *CNC* value might differ in complexity regarding this parameter.
- *Control Flow Complexity (CFC)* enables to measure the complexity in terms of the potential transitions after a split depending on its type. The greater the value of the *CFC*, the greater the overall structural complexity of a workflow.

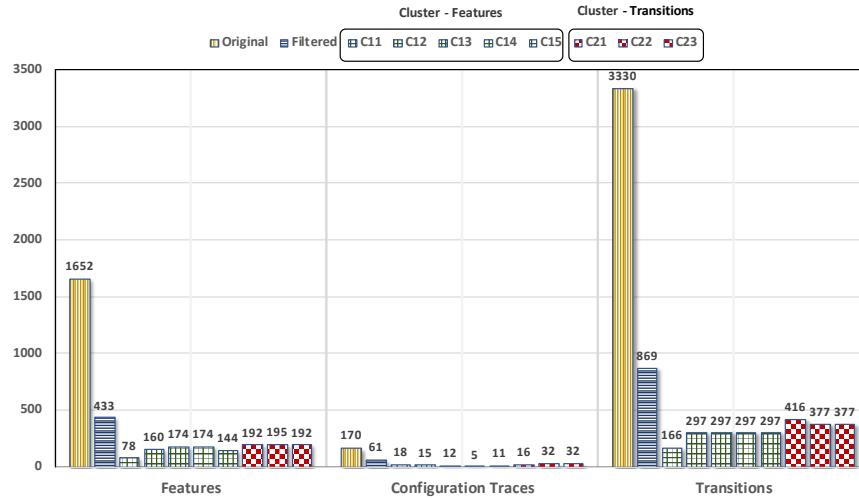
**Figure 8: Characteristics of the configurations logs.**

Table 3 shows the results of these metrics obtained from the discovered configuration workflows. As explained, for every metric the lower value they take, the better the understandability and less complexity. We shall highlight that the metrics associated with the different clusters are aggregated as arithmetic means for a better comparison.

Config. Workflow	Density	CC	CNC	CFC
Original	0,00123	1679	2,016	1677
Filtered	0,00464	437	2,069	434
Cluster-Features	0,01469	125,8	1,892	118,4
Cluster-Transitions	0,00632	118,8	1,213	156

**Table 3: Metrics of the configuration workflows.**

It is interesting to note how configuration workflow for the original has the lowest density in comparison with the others. This is because the number of features is so high and it compensates the largest number of transitions. However, it has the highest complexity compared to CC and CFC. However, no conclusions can be achieved with regard to the complexity based on CNC. Therefore, although the density is the lowest, the other three metrics demonstrate that the workflow for the original is very complex and misunderstood. On the other hand, the workflow for the filtered version has the highest density, thus, it is the less understandable regarding to this metric. However, the complexity shown by the other metrics demonstrates that it is more understandable and less complex than the original configuration workflow.

Comparing the workflow for the filtered version with both cluster versions, we can conclude that both clusters are more understandable due to lower values related to the four complexity metrics, (i.e., density, CC, CNC and CFC).

In fact, these four metrics help us to know the complexity and understandability of the configuration workflows from the design perspective and the elements in the model. Nevertheless, these metrics used to measure the quality of the workflow are inconclusive to measure the real usefulness and quality of the discovered workflows applied to the context of the variability management.

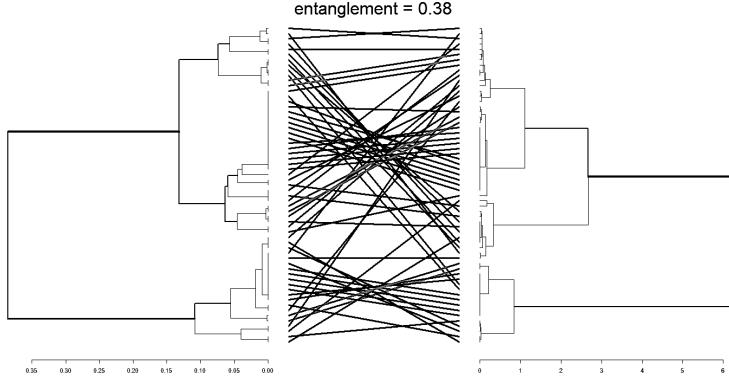
**3.3.2 Analysis of clustering.** As introduced in previous sections, two different entropies are applied to infer the clustering (i.e., *Entropy-features* and *Entropy-transitions*). The two entropy formulas can help to understand the quality of the workflow. Thus, a lower value of entropy more quality of the cluster, hence, workflow has more quality. In this regard, Table 4 gives the values regarding the number of clusters and the entropy for each configuration workflow. In this case, the metrics associated with the clusters are aggregated as arithmetic means for better comparison. It is important to highlight that the clusters group a less number of configuration traces, the entropy of the clusters (as mean) are less in both cases than the original and the filtered solution.

Config. Workflow	N. of Clusters	Entropy features	Entropy transitions	Quality ( $\Delta_{\equiv}$ )
Original	1	1	0,021	1598,84
Filtered	1	1	0,027	365,18
Cluster-Features	5	0,188	-	59,02
Cluster-Trans.	3	-	0,0052	115,037

**Table 4: Comparison of the number of clusters, entropy and quality.**

In order to compare the clusters, the entanglement metric is determined as shown in Figure 9. The entanglement indicates the relation between two dendrograms charts, thus, two different distributions of clusters. The range of the entanglement is [0..1]. The entanglement values closer to 0 are better than to 1. Thus, the entanglement helps to understand how similar are the dendograms, thereby, how similar clusters are with the independence of the entropy. In this case, the entanglement is 0,38 which indicates that both clusters are very similar, in other words, the entropy used to perform the clusters reach similar cluster distributions in this case.

As previously mentioned, the uselessness of the quality metrics related to the workflows leads us to define a new custom metric which enables to establish the quality level of the workflow by relating the number of features and their occurrence within the discovered workflow of a cluster. Thus, a metric that enables us



**Figure 9: Comparative of entanglement between clusters (*Entropy-features* (right) and *Entropy-transitions* (left)).**

to measure how spaghetti is the workflow obtained. Our custom quality metric is defined as follows:

- Quality ( $\Delta_{\equiv}$ ) measures the difference between the total number of features and the ratio of the sum of the number of times that a feature is selected for each configuration trace and the number of configuration traces.

Formally, given a workflow based on a set of configuration traces ( $CT$ ) and a set of features (*Features*), the quality can be determined as the following formula:

$$\Delta_{\equiv} = |Features| - \sum_{f \in Workflow} \frac{occurrences(f)}{|CT|} \quad (3)$$

The range of the quality is  $[0..|Features|]$ , the lower value of quality indicates a better configuration workflow. The number of features and configuration traces are group into the more similar workflow, therefore, it brings about that the quality is near to 0.

For instance, using the example in Figure 3 and the *Cluster 2* in Figure 5, the number included in the rectangle of the feature corresponds to the number of times that the feature is selected regarding the configuration traces. Hence, the quality for the *Cluster 2* can be determined applying the formula as follows:

$$\Delta_{\equiv} = 8 - \left( \frac{3}{3} + \frac{3}{3} + \frac{1}{3} + \frac{1}{3} + \frac{3}{3} + \frac{1}{3} + \frac{2}{3} + \frac{2}{3} \right) \approx 2,67 \quad (4)$$

Comparing the quality results in Table 4, the conclusion is that the original configuration workflow obtains the worst quality and the five clusters obtained using entropy-features have the best quality. Thus, the distribution of configuration traces in the five clusters (cf., Cluster-Features in Figure 8) achieves better results than the original, filtered, even another cluster regarding. In conclusion and according to the defined quality, the Cluster-Features are less complex, more understandable and less spaghetti than the other configuration workflows.

### 3.4 COLOSSI implementation

COLOSSI is supported by the implementation of a tool which is composed of the next main components:

- (1) *Configuration log extractor* is a piece of software module which takes a set of raw configuration log (including timestamps) in a semi-structured format and returns a XES file.

- (2) *Configuration log handler* is another piece of software which takes a FM and a XES log as input. First, apply a set of operations over the FM as described in Section 2.3. Then, a data cleaning is carried out over the XES log to get a filtered configuration log. The output of this connector is a new XES log with the filtered configuration log.
- (3) *Cluster generator* is a Python/R module which takes a XES log file which is translated into a matrix. This matrix enables the entropy calculation and based on the analysis of certain parameters and the dendrogram, the number of clusters is determined. Using this information, a hierarchical agglomerative clustering algorithm is applied to determine the clusters. A new XES log file is generated for each cluster that composed the final output of this component.
- (4) *Discovery connector* is a piece of software which gets the XES file logs of each cluster and automatically feed the ProM to discover the process models by means of the *Inductive Miner*. The output of this component is a process model in Petri-net or BPMN format.

All the resources, thus, configuration logs, the XES files, the workflows discovered, the source code of the COLOSSI tool (i.e., git repository), and a Jupyter notebook that are employed in this work are freely available at <sup>2</sup><http://www.idea.us.es/splc2019/>. The notebook is self-explanatory and allows users to work interactively by executing step-by-step instructions to get the clusters.

### 3.5 Threats to validity

Even though, the experiments presented in this paper provide evidences that the solution proposed is valid, there are some assumptions that we made that may affect their validity. In this section, we discuss the different threats to validity that affect the evaluation.

**External validity.** The inputs used for the experiments presented in this paper were either realistic or designed to mimic realistic feature models. However, we do not control the development process and it may have errors and not encode all ERP configurations.

The major threats to the external validity are:

<sup>2</sup>DOI:10.5281/zenodo.3236337

- *Population validity:* the ERP feature model that we used may not represent all ERP realistic products. Note that the model was provided after an anonymisation process. Moreover, the timestamps used to derive the traces were relying on the appearance within the input file without an explicit enumeration. To reduce these threats to validity, we chose a single large model that was used in different studies in literature.
- *Ecological validity:* while external validity, in general, is focused on the generalisation of the results to other contexts (e.g., using other models), the ecological validity if focused on possible errors in the experiment materials and tools used. To avoid as much as possible such threats, we relied on previously existing algorithms to perform the process discovery.

**Internal validity:** concretely, we developed several metrics that reveals different properties of the workflows, however, there might be characteristics of such workflows that are not revealed.

## 4 RELATED WORK

In this section, we go through the related work of this research. **Configuration workflows.** A formal description of configuration workflows is given in [27]. However, a configuration workflow is a bit different from our definition. An activity of the configuration workflow can be mapped to more than just a feature as in our case. However, our approach is complementary because in the handling process we can group different features as well. Furthermore, although formal semantics and automated support for configuration workflows is presented, no automated mechanism is developed to automatically generate configuration workflows from existing configuration logs. In that sense, our approach complements theirs.

Different possible feature orders are defined in [21]. Those orders are used to build web-based configurators hiding the details of the concrete variability model flavours (e.g., OVM, FMs, CVL, etc.). The orders are built from the structure of the variability model. For instance, in the case of FMs, pre-order, pos-order or in-order can be used to determine the feature order in which features are presented to the user. COLOSSI differs from this approach because we use as input configuration logs to automatically derive and cluster configuration workflows. Our approach can be complementary to [21] because different existing workflows could be also measured using process alignment metrics to determine what's the best feature order to be used.

There exist other approaches [67, 68] focused on the field of product configurator design in which configuration workflows has been tackled from the perspective of machine learning.

**Application of process mining in different contexts.** In order to discover the processes followed by users or systems analysing event logs, process mining has been applied in several scenarios. Depending on the scenario, different are the points of view that could be used to discover a process, such as the activities executed, persons involved, the resources used, the location where the actions occurs, etc. The versatility of process mining techniques has brought about its application to several scenarios [13], being health-care [38, 49, 52] and IT [39, 47, 55] the most active areas.

The case studies where event logs are produced by human behaviour interactions are specially complex, derived from the free

will capacity of the persons that is not always possible to be modelled. This is the context of this paper, where configuration tasks describe the interaction of users with systems. Previous examples in previous scenarios have been developed, such as [2], to analyse how the users interacts with an enterprise resource planning software, or the applicability of software scenarios analysing how the users interact with software to promote improvements about functional specifications or usability aspects [54]. Software development has also provided a complex scenario where process mining can provide mechanism to improve and optimise the known as software process mining [53]. However, configurability issue has not been analysed before with process mining.

**High variability in process mining.** When there is a high human interaction, as in configuration processes, spaghetti and lasagna processes tend to be obtained. The occurrence of infrequent activities or non-repeated sequence of activities in the analysed log events bring about the necessity to apply frequency-based filtering solutions [12] and other based on the discovery of chaotic set of activities that are frequent [60].

The infrequent patterns in process discovery are frequently treated as noise [36], being removed from the log traces to discover a process that represent the most frequent behaviour [56]. Different types of filtering can be performed: (i) filtering the events that are not belong to the mainstream behaviour [12, 56]; (ii) integrating the filtering as a part of the discovery [34, 41, 65, 70]; (iii) filtering traces, in an unsupervised [23] or supervised way [11], and; (iv) including a previous steps for clustering the problem, facilitating the discrimination of traces according to different points of view or dividing different types of behaviour [15, 59].

In summary, up to our knowledge, this is the first solution for workflow retrieval in SPL-related contexts. It is also relevant the use of process mining techniques in new domains. This paper aims at promoting synergies between these two areas of study.

## 5 CONCLUDING REMARKS & LESSONS LEARNED

In this paper, we have coped with the problem of extracting the actual workflows used by SPL configurators analysing configuration logs. To discover configuration workflows, we decided to rely on process mining techniques. Moreover, we propose to apply clustering to improve the resulting configuration workflows reducing the complexity and improving their understandability. From our research on configuration workflows, we learned the following important lessons:

- (1) **Reduce the complexity of the configuration workflows.** We have defined a mechanism based on clustering to divide the configuration logs into smaller configurations groups to facilitate the understanding of the configuration workflows inferred from configuration logs.
- (2) **Quality measurement.** We have defined a set of metrics adapted from business process literature, to measure the quality of the obtained clusters and configuration workflows.
- (3) **Improving decisions about configurators.** The clustering creation and the analysis provide information to expert users about the features that used to be configured together

or sequential lists of features that could be integrated in a single feature.

In future work, we plan to develop new variability-oriented metrics that can show the impact of the numbers of features within the workflows, trying to incorporate characteristics of the feature models into the clustering and process discovery. Moreover, we would like to apply this technique to more scenarios and datasets to complement the validation of our proposal, including in the analysis other methods to tackle spaghetti processes. Further, we consider interesting to investigate a proper way to obtain the best distribution clusters automatically for a defined numbers of clusters. From our point of view, it is also relevant to propose multiple uses of the resulting workflows to help in different areas such as reverse engineering or SPL testing.

## ACKNOWLEDGEMENT

This work has been partially funded by the Ministry of Science and Technology of Spain through ECLIPSE (RTI2018-094283-B-C33), the Junta de Andalucía via the PIRAMIDE and METAMORFOSIS projects, the European Regional Development Fund (ERDF/FEDER), and the MINECO Juan de la Cierva postdoctoral program. The authors would like to thank the Cátedra de Telefónica “Inteligencia en la Red” of the Universidad de Sevilla for its support.

## REFERENCES

- [1] 2016. IEEE Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams. *IEEE Std 1849-2016* (Nov 2016), 1–50. <https://doi.org/10.1109/IEEESTD.2016.7740858>
- [2] Saulius Astromskis, Andrea Janes, and Michael Mairegger. 2015. A Process Mining Approach to Measure How Users Interact with Software: An Industrial Case Study. In *Proceedings of the 2015 International Conference on Software and System Process (ICSSP 2015)*. ACM, New York, NY, USA, 137–141. <https://doi.org/10.1145/2785592.2785612>
- [3] A. Augusto, R. Conforti, M. Dumas, M. L. Rosa, F. M. Maggi, A. Marrella, M. Melcella, and A. Soo. 2019. Automated Discovery of Process Models from Event Logs: Review and Benchmark. *IEEE Transactions on Knowledge and Data Engineering* 31, 4 (April 2019), 686–705. <https://doi.org/10.1109/TKDE.2018.2841877>
- [4] Frank B Baker and Lawrence J Hubert. 1975. Measuring the power of hierarchical cluster analysis. *J. Amer. Statist. Assoc.* 70, 349 (1975), 31–38.
- [5] Geoffrey H Ball and David J Hall. 1965. *ISODATA, a novel method of data analysis and pattern classification*. Technical Report. Stanford research inst Menlo Park CA.
- [6] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later. *Information Systems* 35, 6 (2010), 615–636.
- [7] David Benavides, Pablo Trinidad, Antonio Ruiz Cortés, and Sergio Segura. 2013. *FaMa*. Springer Berlin Heidelberg, Chapter FaMa, 163–171. [https://doi.org/10.1007/978-3-642-36583-6\\_11](https://doi.org/10.1007/978-3-642-36583-6_11)
- [8] Jan Bosch. 2018. The Three Layer Product Model: An Alternative View on SPLs and Variability. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS 2018, Madrid, Spain, February 7–9, 2018*, 1. <https://doi.org/10.1145/3168365.3168366>
- [9] Tadeusz Calinski and Jerzy Harabasz. 1974. A dendrite method for cluster analysis. *Communications in Statistics-theory and Methods* 3, 1 (1974), 1–27.
- [10] Jorge Cardoso. 2005. Control-flow complexity measurement of processes and Weyuker’s properties. In *6th International Enformatika Conference*, Vol. 8. 213–218.
- [11] Hsin-Jung Cheng and Akhil Kumar. 2015. Process mining on noisy logs - Can log sanitization help to improve performance? *Decision Support Systems* 79 (2015), 138–149. <https://doi.org/10.1016/j.dss.2015.08.003>
- [12] Raffaele Conforti, Marcello La Rosa, and Arthur H. M. ter Hofstede. 2017. Filtering Out Infrequent Behavior from Business Process Event Logs. *IEEE Trans. Knowl. Data Eng.* 29, 2 (2017), 300–314. <https://doi.org/10.1109/TKDE.2016.2614680>
- [13] Dusanka Dakic, Darko Stefanovic, Ilija Cosic, Teodora Lolic, and Milovan Medojevic. 2018. BUSINESS APPLICATION: A LITERATURE REVIEW. In *29TH DAAAM INTERNATIONAL SYMPOSIUM ON INTELLIGENT MANUFACTURING AND AUTOMATION*. <https://doi.org/10.2507/29th.daaam.proceedings.125>
- [14] David L Davies and Donald W Bouldin. 1979. A cluster separation measure. *IEEE transactions on pattern analysis and machine intelligence* 2 (1979), 224–227.
- [15] Massimiliano de Leoni, Wil M. P. van der Aalst, and Marcus Dees. 2016. A general framework for correlating, predicting and clustering dynamic behavior based on event logs. *Inf. Syst.* 56 (2016), 235–257. <https://doi.org/10.1016/j.is.2015.07.003>
- [16] Richard A Duda, Peter E Hart, et al. 1973. *Pattern classification and scene analysis*. Vol. 3. Wiley New York.
- [17] Joseph C Dunn. 1974. Well-separated clusters and optimal fuzzy partitions. *Journal of cybernetics* 4, 1 (1974), 95–104.
- [18] A. Durán, D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. 2017. FLAME: a formal framework for the automated analysis of software product lines validated by automated specification testing. *SOSYM* 16, 4 (2017), 1049–1082. <https://doi.org/10.1007/s10270-015-0503-z>
- [19] Alexander Felfernig, Lothar Hotz, Claire Bagley, and Juha Tiihonen. 2014. *Knowledge-Based Configuration*.
- [20] T Frey and H Van Groenewoud. 1972. A cluster analysis of the D2 matrix of white spruce stands in Saskatchewan based on the maximum-minimum principle. *The Journal of Ecology* (1972), 873–886.
- [21] J.A. Galindo, D Dhungana, R Rabiser, D Benavides, G Botterweck, and P. Grünbacher. 2015. Supporting distributed product configuration by integrating heterogeneous variability modeling approaches. *Information and Software Technology* 62, 1 (2015), 78–100.
- [22] José A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2018. Automated analysis of feature models: Quo vadis? *Computing* (11 Aug 2018). <https://doi.org/10.1007/s00607-018-0646-1>
- [23] Lucantonio Ghionna, Gianluigi Greco, Antonella Guzzo, and Luigi Pontieri. 2008. Outlier Detection Techniques for Applications. In *Foundations of Intelligent Systems*, Aijun An, Stan Matwin, Zbigniew W. Raś, and Dominik Ślęzak (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 150–159.
- [24] Maria Halkidi, Michalis Vazirgiannis, and Yannis Batistakis. 2000. Quality scheme assessment in the clustering process. In *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 265–276.
- [25] John A Hartigan. 1975. Clustering algorithms. (1975).
- [26] B. F. A. Hompes, J. C. A. M. Buijs, Wil M. P. van der Aalst, P. M. Dixit, and J. Buurman. 2017. Detecting Changes in Process Behavior Using Comparative Case Clustering. In *Data-Driven Process Discovery and Analysis*, Paolo Ceravolo and Stefanie Rinderle-Ma (Eds.). Springer International Publishing, 54–75.
- [27] Arnaud Hubaux, Andreas Classen, and Patrick Heymans. 2009. Formal Modelling of Feature Configuration Workflows. In *Proceedings of the 13th International Software Product Line Conference (SPLC ’09)*. Carnegie Mellon University, Pittsburgh, PA, USA, 221–230. <http://dl.acm.org/citation.cfm?id=1753235.1753266>
- [28] A Hubaux, P b Heymans, P.-Y Schobbens, D Deridder, and E.K. Abbasi. 2013. Supporting multiple perspectives in feature-based configuration. *SOSYM* 12, 3 (2013), 641–663. <https://doi.org/10.1007/s10270-011-0220-1>
- [29] Lawrence J Hubert and Joel R Levin. 1976. A general statistical framework for assessing categorical clustering in free recall. *Psychological bulletin* 83, 6 (1976), 1072.
- [30] A. K. Jain, M. N. Murty, and P. J. Flynn. 1999. Data Clustering: A Review. *ACM Comput. Surv.* 31, 3 (Sept. 1999), 264–323. <https://doi.org/10.1145/331499.331504>
- [31] Ari Kobren, Nicholas Monath, Akshay Krishnamurthy, and Andrew McCallum. 2017. A Hierarchical Algorithm for Extreme Clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD ’17)*. ACM, New York, NY, USA, 255–264.
- [32] Wojtek J Krzanowski and YT Lai. 1988. A criterion for determining the number of groups in a data set using sum-of-squares clustering. *Biometrics* (1988), 23–34.
- [33] L Lebart, A Morineau, and M Piron. 2000. *Statistique exploratoire multidimensionnelle*. Dunod, Paris, France. (2000).
- [34] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. 2014. Discovering Block-Structured Process Models from Incomplete Event Logs. In *Petri Nets (Lecture Notes in Computer Science)*, Vol. 8489. Springer, 91–110.
- [35] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. 2015. Scalable Process Discovery with Guarantees. In *Enterprise, Business-Process and Information Systems Modeling*, Khaled Gaaloul, Rainer Schmidt, Selmin Nurcan, Sérgio Guerreiro, and Qin Ma (Eds.). Springer International Publishing, Cham, 85–101.
- [36] Linh Thao Ly, Conrad Indiono, Jürgen Mangler, and Stefanie Rinderle-Ma. 2012. Data Transformation and Semantic Log Purging for Process Mining. In *CAiSE (Lecture Notes in Computer Science)*, Vol. 7328. Springer, 238–253.
- [37] David J. C. MacKay. 2002. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA.
- [38] R. S. Mans, M. H. Schonenberg, M. Song, W. M. P. van der Aalst, and P. J. M. Bakker. 2009. Application of Process Mining in Healthcare – A Case Study in a Dutch Hospital. In *Biomedical Engineering Systems and Technologies*, Ana Fred, Joaquim Filipe, and Hugo Gamboa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 425–438.
- [39] Laura Măruster and Nick R. T. P. van Beest. 2009. Redesigning business processes: a methodology based on simulation and techniques. *Knowledge and Information Systems* 21, 3 (25 Jun 2009), 267. <https://doi.org/10.1007/s10115-009-0224-0>
- [40] Laura Măruster, A. J. M. M. Weijters, Wil M. P. van der Aalst, and Antal van den Bosch. 2002. Discovering Direct Successors in Process Logs. In *Discovery Science*,

- 5th International Conference, DS 2002, Lübeck, Germany, November 24–26, 2002, Proceedings.* 364–373. [https://doi.org/10.1007/3-540-36182-0\\_37](https://doi.org/10.1007/3-540-36182-0_37)
- [41] Laura Maruster, A. J. M. M. Weijters, Wil M. P. van der Aalst, and Antal van den Bosch. 2006. A Rule-Based Approach for Process Discovery: Dealing with Noise and Imbalance in Process Logs. *Data Min. Knowl. Discov.* 13, 1 (2006), 67–87.
- [42] John O McClain and Vithala R Rao. 1975. Clustisz: A program to test for the quality of clustering of a set of objects. *JMR, Journal of Marketing Research (pre-1986)* 12, 000004 (1975), 456.
- [43] Jan Mendling. 2008. *Metrics for Business Process Models*. Springer Berlin Heidelberg, Berlin, Heidelberg, 103–133. [https://doi.org/10.1007/978-3-540-89224-3\\_4](https://doi.org/10.1007/978-3-540-89224-3_4)
- [44] Glenn W Milligan. 1980. An examination of the effect of six types of error perturbation on fifteen clustering algorithms. *Psychometrika* 45, 3 (1980), 325–342.
- [45] Glenn W Milligan. 1981. A monte carlo study of thirty internal criterion measures for cluster analysis. *Psychometrika* 46, 2 (1981), 187–199.
- [46] Juliana Alves Pereira, Paweł Matuszyk, Sebastian Krieter, Myra Spiliopoulou, and Gunter Saake. 2018. Personalized recommender systems for product-line configuration processes. *Computer Languages, Systems & Structures* 54 (2018), 451–471. <https://doi.org/10.1016/j.cl.2018.01.003>
- [47] José Miguel Pérez-Álvarez, Alejandro Maté, María Teresa Gómez López, and Juan Trujillo. 2018. Tactical Business-Process-Decision Support based on KPIs Monitoring and Validation. *Computers in Industry* 102 (2018), 23–39.
- [48] Ricardo Pérez-Castillo, María Fernández-Ropero, and Mario Piatti. 2019. Business process model refactoring applying IBUPROFEN. An industrial evaluation. *Journal of Systems and Software* 147 (2019), 86 – 103. <https://doi.org/10.1016/j.jss.2018.10.012>
- [49] Luca Perimal-Lewis, David Teubner, Paul Hakendorf, and Chris Horwood. 2016. Application of process mining to assess the data quality of routinely collected time-based performance data sourced from electronic health records by validating process conformance. *Health informatics journal* 22 4 (2016), 1017–1029.
- [50] DA Ratkowsky and GN Lance. 1978. Criterion for determining the number of groups in a classification. (1978).
- [51] F James Rohlf. 1974. Methods of comparing classifications. *Annual Review of Ecology and Systematics* 5, 1 (1974), 101–113.
- [52] Anne Rozinat, Ivo S. M. de Jong, Christian W. Günther, and Wil M. P. van der Aalst. 2009. Process Mining Applied to the Test Process of Wafer Scanners in ASML. *IEEE Trans. Systems, Man, and Cybernetics, Part C* 39, 4 (2009), 474–479.
- [53] Vladimir Rubin, Christian W. Günther, Wil M. P. van der Aalst, Ekkart Kindler, Boulewijn F. van Dongen, and Wilhelm Schäfer. 2007. Process Mining Framework for Software Processes. In *Software Process Dynamics and Agility*, Qing Wang, Dietmar Pfahl, and David M. Raffo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–181.
- [54] Vladimir A. Rubin, Alexey A. Mitsyuk, Irina A. Lomazova, and Wil M. P. van der Aalst. 2014. Process Mining Can Be Applied to Software Too!. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14)*. ACM, New York, NY, USA, Article 57, 8 pages. <https://doi.org/10.1145/2652524.2652583>
- [55] Mahdi Sahlabadi, Ravie Chandren Muniyandi, and Zarina Shukur. 2014. Detecting abnormal behavior in social network websites by using a process mining technique. *Journal of Computer Science* 10, 3 (2014), 393–402. <https://doi.org/10.3844/jcsp.2014.393.402>
- [56] Mohammadreza Fani Sani, Sebastiaan J. van Zelst, and Wil M. P. van der Aalst. 2017. Improving Process Discovery Results by Filtering Outliers Using Conditional Behavioural Probabilities. In *Business Process Management Workshops - BPM 2017 International Workshops, Barcelona, Spain, September 10–11, 2017, Revised Papers*. 216–229. [https://doi.org/10.1007/978-3-319-74030-0\\_16](https://doi.org/10.1007/978-3-319-74030-0_16)
- [57] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic semantics of feature diagrams. *Computer Networks* 51, 2 (2007), 456–479. <https://doi.org/10.1016/j.comnet.2006.08.008>
- [58] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2010. The Variability Model of The Linux Kernel.. In *VAMOS*, Vol. 10. 45–51.
- [59] Minseok Song, Christian W. Günther, and Wil M. P. van der Aalst. 2009. Trace Clustering in. In *Business Process Management Workshops*, Danilo Ardagna, Massimo Mecella, and Jian Yang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 109–120.
- [60] Niek Tax, Natalia Sidorova, and Wil M. P. van der Aalst. 2019. Discovering more precise process models from event logs by filtering out chaotic activities. *J. Intell. Inf. Syst.* 52, 1 (2019), 107–139. <https://doi.org/10.1007/s10844-018-0507-6>
- [61] T Thüm, S Apel, C Kästner, I Schaefer, and G.A Saake. 2014. A classification and survey of analysis strategies for software product lines. *ACMCS* 47, 1 (2014). <https://doi.org/10.1145/2580950>
- [62] Wil M. P. van der Aalst. 2011. *Analyzing “Spaghetti Processes”*. Springer Berlin Heidelberg, Berlin, Heidelberg, 301–317 pages. [https://doi.org/10.1007/978-3-642-19345-3\\_12](https://doi.org/10.1007/978-3-642-19345-3_12)
- [63] Wil M. P. van der Aalst. 2016. *Process Mining - Data Science in Action, Second Edition*. Springer.
- [64] Boulewijn F. van Dongen, Ana Karla A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and Wil M. P. van der Aalst. 2005. The ProM Framework: A New Era in Process Mining Tool Support. In *Applications and Theory of Petri Nets 2005, 26th International Conference, ICATPN 2005, Miami, USA, June 20–25, 2005, Proceedings*. 444–454. [https://doi.org/10.1007/11494744\\_25](https://doi.org/10.1007/11494744_25)
- [65] Seppe K. L. M. vanden Broucke and Jochen De Weerdt. 2017. Fodina: A robust and flexible heuristic process discovery technique. *Decision Support Systems* 100 (2017), 109–118. <https://doi.org/10.1016/j.dss.2017.04.005>
- [66] Angel Jesus Varela-Vaca and Rafael M. Gasca. 2013. Towards the automatic and optimal selection of risk treatments for business processes using a constraint programming approach. *Information & Software Technology* 55, 11 (2013), 1948–1973.
- [67] Yue Wang and Mitchell Tseng. 2014. Attribute selection for product configurator design based on Gini index. *International Journal of Production Research* 52, 20 (2014), 6136–6145. <https://doi.org/10.1080/00207543.2014.917216>
- [68] Yue Wang and Mitchell M. Tseng. 2011. Adaptive attribute selection for configurator design via Shapley value. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 25, 2 (2011), 185–195. <https://doi.org/10.1017/S0890060410000624>
- [69] Joe H Ward Jr. 1963. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association* 58, 301 (1963), 236–244.
- [70] A. J. M. M. Weijters and J. T. S. Ribeiro. 2011. Flexible Heuristics Miner (FHM). In *CIDM*. IEEE, 310–317.

# Towards Quality Assurance of Software Product Lines with Adversarial Configurations\*

Paul Temple

NaDI, PReCISE, Faculty of Computer  
Science, University of Namur  
Namur, Belgium

Battista Biggio

University of Cagliari  
Cagliari, Italy

Mathieu Acher

Univ Rennes, IRISA, Inria, CNRS  
Rennes, France

Jean-Marc Jézéquel

Univ Rennes, IRISA, Inria, CNRS  
Rennes, France

Gilles Perrouin

NaDI, PReCISE, Faculty of Computer  
Science, University of Namur  
Namur, Belgium

Fabio Roli

University of Cagliari  
Cagliari, Italy

## ABSTRACT

Software product line (SPL) engineers put a lot of effort to ensure that, through the setting of a large number of possible configuration options, products are acceptable and well-tailored to customers' needs. Unfortunately, options and their mutual interactions create a huge configuration space which is intractable to exhaustively explore. Instead of testing all products, machine learning is increasingly employed to approximate the set of acceptable products out of a small training sample of configurations. Machine learning (ML) techniques can refine a software product line through learned constraints and *a priori* prevent non-acceptable products to be derived. In this paper, we use adversarial ML techniques to generate *adversarial configurations* fooling ML classifiers and pinpoint incorrect classifications of products (videos) derived from an industrial video generator. Our attacks yield (up to) a 100% misclassification rate and a drop in accuracy of 5%. We discuss the implications these results have on SPL quality assurance.

## CCS CONCEPTS

- Software and its engineering → Software product lines.

## KEYWORDS

software product line; software variability; software testing; machine learning; quality assurance

### ACM Reference Format:

Paul Temple, Mathieu Acher, Gilles Perrouin, Battista Biggio, Jean-Marc Jézéquel, and Fabio Roli. 2019. Towards Quality Assurance of Software Product Lines with Adversarial Configurations. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3336294.3336309>

\*Gilles Perrouin is an FNRS Research Associate. This research was partly supported by EOS Verilearn project grant no. O05518F-RG03. This research was also funded by the ANR-17-CE25-0010-01 VaryVary project.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336309>

## 1 INTRODUCTION

Testers don't like to break things; they like to dispel the illusion that things work. [33]

Software Product Line Engineering (SPLE) aims at delivering *massively customized* products within shortened development cycles [18, 47]. To achieve this goal, SPLE systematically reuses software assets realizing the functionality of one or more *features*, which we loosely define as units of variability. Users can specify products matching their needs by selecting/deselecting the features and provide additional values for their attributes. Based on such *configurations*, the corresponding products can be obtained as a result of the product derivation phase. A long-standing issue for developers and product managers is to gain confidence that all possible products are functionally viable, e.g., all products compile and run. This is a hard problem, since modern software product lines (SPLs) can involve thousands of features and practitioners cannot test all possible configurations and corresponding products due to combinatorial explosion. Research efforts rely on variability models (e.g., feature diagrams) and solvers (SAT, CSP, SMT) to compactly define how features can and cannot be combined [2, 4, 5, 49]. Together with advances in model-checking, software testing and program analysis techniques, it is conceivable to assess the functional validity of configurations and their associated combination of assets within a product of the SPL [12, 13, 17, 39, 54, 57].

Yet, when dealing with qualities on the derived products (performance, costs, etc.), several unanswered challenges remain from the specification of feature-aware quantities to the best trade-offs between products and family-based analyses (e.g., [36, 59]). In our industrial case-study, the MOTIV video generator [26], one can approximately generate  $10^{314}$  video variants. Furthermore, it takes about 30 minutes to create a new video: a non-acceptable (e.g., a too noisy or dark) video can lead to a tremendous waste of resources. A promising approach is to sample a number of configurations and predict the quantitative or qualitative properties of the remaining configurations using Machine Learning (ML) techniques [29, 42, 48, 51, 52, 56, 58]. These techniques create a predictive model (a classifier) from such sampled configurations and infer the properties of yet unseen configurations with respect to their distribution's similarity. This way, unseen configurations that do not match specific properties can be automatically discarded and constraints can be added to the feature diagram in order to avoid them permanently [55, 56]. However, we need to trust the

ML classifier [1, 41] to avoid costly misclassifications. In the ML community, it has been demonstrated that some forged instances, called *adversarial*, can fool a given classifier [11]. *Adversarial machine learning* (advML) thus refers to techniques designed to fool (e.g., [6, 7, 41]), evaluate the security (e.g., [9]) and even improve the quality of learned classifiers [27]. Our overall goal is to study how advML techniques can be used to assess quality assurance of ML classifiers employed in SPL activities. In this paper, we design a generator of adversarial configurations for SPLs and measure how the prediction ability of the classifier is affected by such *adversarial* configurations. We also provide scenarios of usage of advML for quality assurance of SPLs. We discuss how adversarial configurations raise questions about the quality of the variability model or the testing oracle of SPL's products. This paper makes the following contributions:

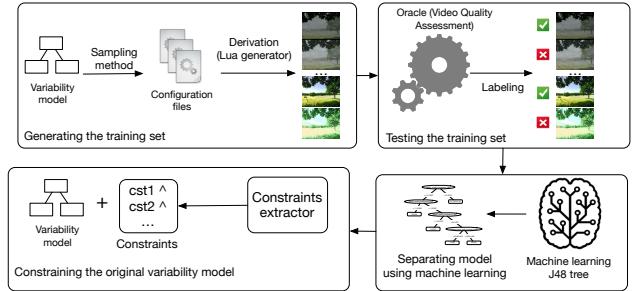
- (1) An adversarial attack generator, based on evasions attacks and dedicated to SPLs;
- (2) An assessment of its effectiveness and a comparison against a random strategy, showing that up to 100% of the attacks are valid with respect to the variability model and successful in fooling the prediction of acceptable/non-acceptable videos, leading to a 5% accuracy loss;
- (3) A qualitative discussion on the generated adversarial configurations w.r.t. to the classifier training set, its potential improvement and the practical impact of advML in the quality assurance workflow of SPLs.
- (4) The public availability of our implementation and empirical results at [https://github.com/templep/SPLC\\_2019](https://github.com/templep/SPLC_2019)

The rest of this paper is organized as follows: Section 2 presents the case study and gives background information about ML and advML; Section 3 shows how advML is used in the context of MOTIV; Section 4 describes experimental procedures and discusses results; Section 5 and 6 present some potential threats that could mitigate our conclusions and propose qualitative discussions about how adversarial configurations could be leveraged for SPLs developers. Section 7 covers related work and Section 8 wraps up the paper with conclusions.

## 2 BACKGROUND

### 2.1 Motivating case: MOTIV generator

MOTIV is an industrial video generator which purpose is to provide synthetic videos that can be used to benchmark computer vision based systems. Video sequences are generated out of configurations specifying the content of the scenes to render [56]. MOTIV relies on a variability model that documents possible values of more than 100 configuration options, each of them affecting the *perception* of generated videos and the achievement of subsequent tasks, such as recognizing moving objects. Perception's variability relates to changes in the background (e.g., being a forest or buildings), objects passing in front of the camera (with varying distances to the camera and different trajectories), blur, etc. There are 20 Boolean options, 46 categorical (encoded as enumerations) options (e.g., to use predefined trajectories) and 42 real-value options (e.g., dealing with blur or noise). Precisely, in average, enumerations contain about 7 elements each and real-value options vary between 0 and 27.64 with a precision of  $10^{-5}$ . Excluding (very few) constraints in



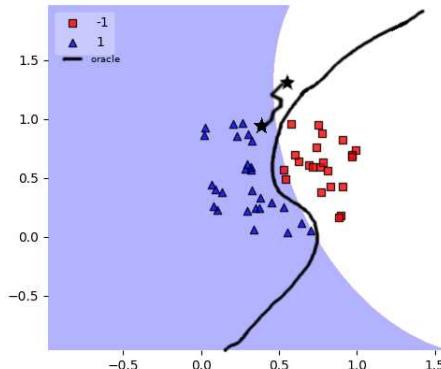
**Figure 1: Refining the variability model of MOTIV video generator via an ML classifier.**

the variability model, we over-estimate the video variants' space size:  $2^{20} * 7^{46} * ((0 - 27.64) * 10^5)^{42} \approx 10^{314}$ . Concretely, MOTIV takes as input a text file describing the scene to be captured by a synthetic camera as well as recording conditions. Then, Lua [32] scripts are called to compose the scene and apply desired visual effects resulting in a video sequence. To realize variability, the Lua code use parameters in functions to activate or deactivate options and to take into account values (enumerations or real values) defined into the configuration file. A highly challenging problem is to identify feature values and interactions that make the identification of moving objects extremely difficult if not impossible. Typically, some of the generated videos contain too much noise or blur. In other words, they are *not acceptable* as they cannot be used to benchmark object tracking techniques. Another class of non-acceptable videos is composed of the ones in which pixels value do not change, resulting in a succession of images for which all pixels have the same color: nothing can be perceived. As mentioned in Section 1, non-acceptable videos represent a waste of time and resources: 30 minutes of CPU-intensive computations per video, not including to run benchmarks related to object tracking (several minutes depending on the computer vision algorithm). We therefore need to constraint our variability model to avoid such cases.

### 2.2 Previous work: ML and MOTIV

We previously used ML classification techniques to predict the acceptability of unseen video variants [56]. We summarise this process in Figure 1.

We first sample valid configurations using a random strategy (see Temple *et al.* [56] for details) and generate the associated video sequences. A computer program playing the role of a *testing oracle* labels videos as acceptable (in green) or non-acceptable (in red). This oracle implements image quality assessment [23] defined by the authors via an analysis of frequency distribution given by Fourier transformations. An ML classifier (in our case, a decision tree) can be trained on such labelled videos. “Paths” (traversals from the top to the leaves) leading to non-acceptable videos can easily be transformed into new constraints and injected in the variability model. An ML classifier can make errors, preventing acceptable videos (false negatives) or allowing non-acceptable videos (false positives). Most of these errors can be attributed to the confidence of the classifier coming from both its design (*i.e.*, the set of approximations used to build its decision model) and the training



**Figure 2: Adversarial configurations (stars) are at the limit of the separating function learned by the ML classifier**

set (and more specifically the distribution of the classes). Areas of low confidence exist if configurations are very dissimilar to those already seen or at the frontier between two classes. We use advML to quantify these errors and their impact on MOTIV.

### 2.3 ML and advML

*ML classification.* Formally, a classification algorithm builds a function  $f : X \mapsto Y$  that associates a label in the set of predefined classes  $y \in Y$  with configurations represented in a feature space (noted  $x \in X$ ). In MOTIV, only two classes are defined:  $Y = \{-1, +1\}$ , respectively representing acceptable and non-acceptable videos.  $X$  represents a set of configurations and the configuration space is defined by configuration options of the underlying feature model (and their definition domain). The classifier  $f$  is trained on a data set  $D$  constituted of a set of pairs  $(x_i^t, y_i^t)$  where  $x^t \in X$  is a set of valid configurations from the variability model and  $y^t \in Y$  their associated labels. To label configurations in  $D$ , we use an oracle (see Figure 1). Once the classifier is trained,  $f$  induces a separation in the feature space (shown as the transition from the blue/left to the white/right area in Figure 2) that mimics the oracle: when an unseen configuration occurs, the classifier determines instantly in which class this configuration belongs to. Unfortunately, the separation can make prediction errors since the classifier is based on statistical assumptions and a (small) training sample. We can see in Figure 2 that the separation diverges from the solid black line representing the target oracle. As a result, two squares are misclassified as being triangles. Classification algorithms realise trade-offs between the necessity to classify the labelled data correctly, taking into account the fact that it can be noisy or biased and its ability to generalise to unseen data. Such trade-offs lead to approximations that can be leveraged by adversarial configurations (shown as stars in Figure 2).

*AdvML and evasion attacks.* According to Biggio *et al.* [11], deliberately attacking an ML classifier with crafted malicious inputs was proposed in 2004. Today, it is called adversarial machine learning and can be seen as a sub-discipline of machine learning. Depending on the attackers' access to various aspects of the ML system (dataset, ability to update the training set) and their goals, various kinds of attacks [6–10] are available: they are organised in a taxonomy [1, 11]. In this paper, we focus on *evasion attacks*: these attacks move labelled data to the other side of the separation (putting it in the opposite class) via successive modifications of features' values.

Since areas close to the separation are of low confidence, such adversarial configurations can have a significant impact if added to the training set. To determine the direction to move the data towards the separation, a gradient-based method has been proposed by Biggio *et al.* [6]. This method requires the attacked ML algorithm to be differentiable. One of such differentiable classifiers is the Support Vector Machine (SVM), parameterizable with a kernel function<sup>1</sup>.

## 3 EVASION ATTACKS FOR MOTIV

### 3.1 A dedicated Evasion Algorithm

---

**Algorithm 1** Our algorithm conducting the gradient-descent evasion attack inspired by [6]

---

**Input:**  $x^0$ , the initial configuration;  $t$ , the step size;  $nb\_disp$ , the number of displacements;  $g$ , the discriminant function  
**Output:**  $x^*$ , the final attack point

- (1)  $m = 0$ ;
- (2) Set  $x^0$  to a copy of a configuration of the class from which the attack starts;
- while**  $m < nb\_disp$  **do**
- (3)  $m = m+1$ ;
- (4) Let  $\nabla F(x^{m-1})$  a unit vector, normalisation of  $\nabla g(x^{m-1})$ ;
- (5)  $x^m = x^{m-1} - t \nabla F(x^{m-1})$ ;
- end while**
- (6) return  $x^* = x^m$ ;

---

Algorithm 1 presents our adaptation of Biggio *et al.*'s evasion attack [6]. First, we select an initial configuration to be moved ( $x^0$ ): selection trade-offs are discussed in the next section. Then, we need to set the step size ( $t$ ), a parameter controlling the convergence of the algorithm. Large steps induce difficulties to converge, while small steps may trap the algorithm in a local optimum. While the original algorithm introduced a termination criterion based on the impact of the attack on the classifier between each move (if this impact was smaller than a threshold  $\epsilon$ , the algorithm stopped; assuming an optimal attack) we fixed the maximal number of displacements  $nb\_disp$  in advance. This allows for a controllable computation budget, as we observed that for small step sizes the number of displacements required to meet the termination criterion was too large. The function  $g$  is the discriminant function and is defined by the ML algorithm that is used. It is defined as  $g : X \mapsto \mathbb{R}$  that maps a configuration to a real number. In fact, only the sign of  $g$  is used to assign a label to a configuration  $x$ . Thus,  $f : X \mapsto Y$  can be decomposed in two successive functions: first  $g : X \mapsto \mathbb{R}$  that maps a configuration to a real value and then  $h : \mathbb{R} \mapsto Y$  with  $h = sign(g)$ . However,  $|g(x)|$  (the absolute value of  $g$ ) intuitively reflects the confidence the classifier has in its assignment of  $x$ .  $|g(x)|$  increases when  $x$  is far from the separation and surrounded by other configurations from the same class and is smaller when  $x$  is close to the separation. The term discriminant function has been used by Biggio *et al.* [6] and should not be confused with the unrelated discriminator component of GANs by Goodfellow *et al.* [27]. In GANs, the discriminator is part of the "robustification process". It is an ML classifier striving to determine whether an input has

<sup>1</sup>most common functions are linear, radial based functions and polynomial

been artificially produced by the other GANs' component, called the generator. Its responses are then exploited by the generator to produce increasingly realistic inputs. In this work, we only generate adversarial configurations, though GANs are envisioned as follow-up work.

Concretely, the core of the algorithm consists of the *while* loop that iterates over the number of displacements. Statement (4) determines the direction towards the area of maximum impact with respect to the classifier (explaining why only a unit vector is needed).  $\nabla g(x^{m-1})$  is the gradient of  $g(x^{m-1})$  and the direction of interest towards which the adversarial configuration should move. This vector is then multiplied by the step size  $t$  and subtracted to the previous move (5). The final position is returned after the number of displacements has been reached. For statements (4) and (5) we simplified the initial algorithm [6]: we do not try to mimic as much as possible existing configurations as we look forward to some diversity. In an open ended feature space, gradient can grow indefinitely possibly preventing the algorithm to terminate. Biggio *et al.* [6] set a maximal distance representing a boundary of the feasible region to keep the exploration under control. In MOTIV, this boundary is represented by the hard constraints in the variability model. Because of the heterogeneity of MOTIV features, cross-tree constraints and domain values are difficult to specify and enforce in the attack algorithm. SAT/SMT solvers would slow down the attack process. We only take care of the type of feature values (natural integers, floats, Boolean). For example, we reset to zero natural integer values that could be negative due to displacements or we ensure that Boolean values are either 0 or 1.

As introduced in Section 2, decision trees are not directly compatible with evasion attacks as the underlying mathematical model is highly non-linear making it non-derivable (forbidding to compute a gradient). We learn another classifier (*i.e.*, a Support Vector Machine) on which we can perform evasion attacks directly [6, 11]. We rely upon evidence that attacks conducted on a specific ML model can be transferred to others [14, 20, 21].

### 3.2 Implementation

We implemented the above procedure in Python 3 (scripts available on the companion website). Figure 3 depicts some images of videos generated out of adversarial configurations.

MOTIV's variability model embeds enumerations which are usually encoded via integers. The main difference between the two is the logical order that is inherent to integers but not encoded into enumerations. As a result, some ML techniques have difficulties to deal with them. The solution is to "dummify" enumerations into a set of Boolean features, which truth values take into account exclusion constraints in the original enumerations. Conveniently, Python provides the *get\_dummies* function from the pandas library which takes as input a set of configurations and feature indexes to dummify. For each feature index, the function creates and returns a set of Boolean features representing the literals' indexes encountered while running through given configurations: if the *get\_dummies* function detects values in the integer range [0, 9] for a feature associated to an enumeration, it will return a set of 10 Boolean features representing literals' indexes in that range. It also takes care of preserving the semantics of enumerations. However,

dummification is not without consequences for the ML classifier. First, it increases the number of dimensions: our 46 initial enumerations would be transformed into 145 features. Doing so may expose the ML algorithm to the *curse of dimensionality* [3]: as the number of features increases in the feature space, configurations that look alike (*i.e.*, with close feature values and the same label) tend to go away from each other, making the learning process more complex. This curse has also been recognised to have an impact on SPL activities [19]. Dummification implies that we will operate our attacks in a feature space *essentially different* from the one induced by the real SPL. This means that we need to transpose the generated attacks in the dummified feature space back to the original SPL one, raising one main issue: there is no guarantee that an attack relevant in the dummified space is still efficient in the reduced original space (the separation may simply not be the same). Additionally, gradient methods operate per feature only, meaning that exclusion constraints in dummified enumerations are ignored. That is, when transposed back to the original configuration space, invalid configurations would need to be "fixed", potentially putting these adversarial configurations away from the optimum computed by the gradient method. For all these reasons, we decided to operate on the initial feature space, acknowledging the threat of considering enumerations as ordered. We conducted a preliminary analysis<sup>2</sup> that showed that the order of the importance of the features were kept whether we use a dummified or the initial feature space. So this threat is minor in comparison of the pitfalls of dummification. We do not make any further distinctions between the two terms since we use them without making any transformations.

As mentioned in Section 3.1, we conducted attacks on a support vector machine with a linear kernel since it was faster according to a preliminary experiment. Scripts as well as data used to compare predictions can be found on the companion webpage.

## 4 EVALUATION

### 4.1 Research questions

We address the following research questions:

**RQ1:** *How effective is our adversarial generator to synthesize adversarial configurations?* Effectiveness is measured through the capability of our evasion attack algorithm to generate configurations that are misclassified:

- **RQ1.1:** Can we generate adversarial configurations that are wrongly classified?
- **RQ1.2:** Are all generated adversarial configurations valid w.r.t. constraints in the VM?
- **RQ1.3:** Is using the evasion algorithm more effective than generating adversarial configurations with random modifications?
- **RQ1.4:** Are attacks effective regardless of the targeted class?

**RQ2:** *What is the impact of adding adversarial configurations to the training set regarding the performance of the classifier?* The intuition is that adding adversarial configurations to the training set could improve the performance of the classifier when evaluated on a test set.

<sup>2</sup>available on the companion webpage: [https://github.com/templep/SPLC\\_2019](https://github.com/templep/SPLC_2019)

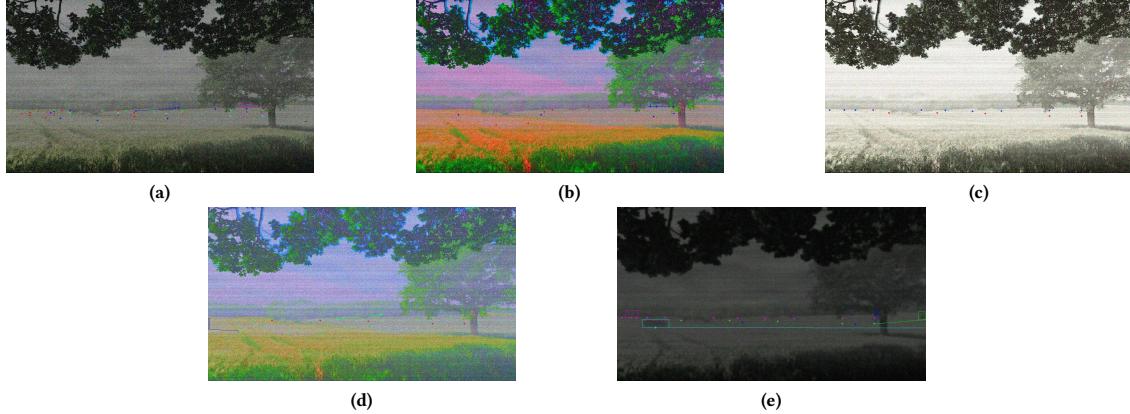


Figure 3: Examples of generated videos using evasion attack

## 4.2 Evaluation protocol

Our evaluation dataset is composed of 4,500 randomly sampled and valid video configurations, that we used in previous work [56]. We selected 500 configurations to train the classifier keeping a similar representation of non-acceptable configurations (10%, *i.e.*,  $\approx 50$  configurations) compared to the whole set. The remaining 4,000 configurations are used as a test set and also have a similar representation regarding acceptable/non-acceptable configurations. This setting contrasts with a common practice of using a high percentage (*i.e.*, around 66%) of available examples to train the classifier. However, due to the low number of non-acceptable configurations, such a setting is impossible.  $k$ -fold cross-validation is another common practice used when few data points are available for training (4,500 configurations is an arguably low number with respect to the size of the variant space). Cross-validation is used to validate/select a classifier when several are created, for instance when trying to fine-tuned hyper-parameters, which is not our case here. Furthermore, separating our 4,500 configurations into smaller sets is likely to create a lot of sets without any non-acceptable configurations. None of these practices seem to be adapted to our case.

The key point is that only about 10% of configurations are non-acceptable. This is a ratio that we cannot control exactly as it depends from the targeted non-functional property. In order to reduce imbalance, several data augmentation techniques exist like SMOTE [16]. Usually, they create artificial configurations while maintaining the configurations' distribution in the feature space. In our case, we compute the centroid between two configurations and use it as a new configuration. Thanks to the centroid method, we can bring perfect balance between the two classes (*i.e.*, 50% of acceptable configurations and non-acceptable configurations). Technically, we compute how many configurations are needed to have perfectly balanced sets (*i.e.*, training and test sets): We select randomly two configurations from the less represented class and compute the centroid between them, check that it is a never-seen-before configuration and adds it to the available configurations. The process is repeated until the number of configurations required is reached. Once a centroid is added to the set of available configurations, it is available as a configuration to create the next centroid.

In the remainder, we present the results with both original and balanced data sets in order to assess whether the impact of class representation imbalance on adversarial attacks. We configured our

evasion attack generator with the following settings: *i*) we set the number of attacks points to generate 4000 configurations for RQ1 and 25 configurations for RQ2 as explained hereafter; *ii*) considered step size ( $t$ ) values are  $\{10^{-6}; 10^{-4}; 10^{-2}; 1; 10^2; 10^4; 10^6\}$ ; *iii*) the number of iterations is fixed to 20, 50 or 100. To mitigate randomness, we repeat ten times the experiments. All results discussed in this paper can also be found on our companion webpage<sup>3</sup>.

## 4.3 Results

### 4.3.1 RQ1: How effective is our adversarial generator to synthesize adversarial configurations?

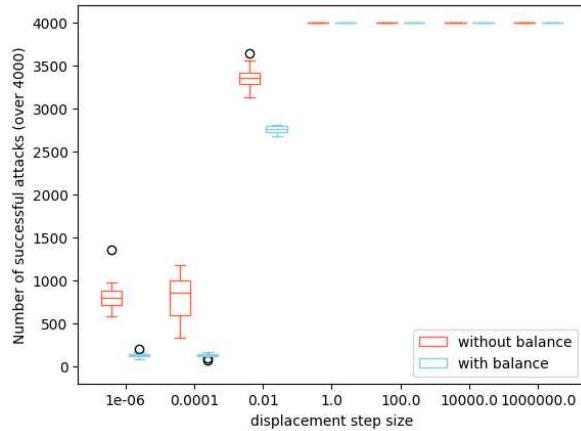
To answer this question, we assess the number of wrongly classified adversarial configurations over 4,000 generations (and about 7,000 configurations when the training set is balanced) and compare them to a random baseline: to the best of our knowledge, there is no comparable evasion attack.

#### RQ1.1: Can we generate adversarial configurations that are wrongly classified?

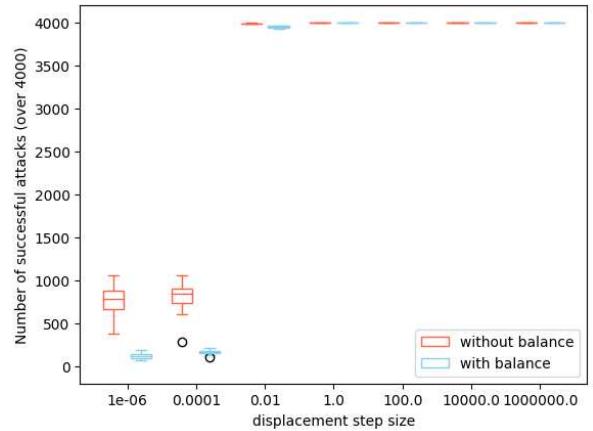
For each run, a newly created adversarial (*i.e.*, after  $nb\_disp$  is reached) configuration is added to the set of initial configurations that can be selected to start an evasion attack. We therefore give a chance to previous adversarial configurations to continue further their displacement towards the global optimum of the gradient.

Figure 4 shows box-plots resulting of ten runs for each attack setting. We also show results when the training set is imbalanced (*i.e.*, using the previous training set containing 500 configurations with about 10% of non-acceptable configurations) and when it is balanced (*i.e.*, increasing the number of non-acceptable configurations using the data augmentation technique described above). Both Figure 4a and Figure 4b indicate that we can always achieve 100% of misclassified configurations with our attacks. Regarding Figure 4a, all generated configurations become misclassified when step size is set to 1.0 or a higher value. When 100 displacements are allowed (see Figure 4b), the limit appears earlier, *i.e.*, when  $t$  equals 0.01. Similar results can be obtained when the number of maximum displacements is set to 50, the only difference is that with  $t$  set to 0.01 not all adversarial configurations are misclassified but about 3,900 (97.5%) when the training set is imbalanced and about 3,700 (92.5%) with a balanced set.

<sup>3</sup>[https://github.com/templep/SPLC\\_2019](https://github.com/templep/SPLC_2019)



(a) Number of misclassified adversarial configurations (20 displacements)



(b) Number of misclassified adversarial configurations (100 displacements)

**Figure 4: Number of successful attacks on class *acceptable*; X-axis represents different step size values  $t$  while Y-axis is the number of misclassified adversarial configurations by the classifier. For each  $t$  value, results with balanced and imbalanced training set are shown (respectively in blue and orange).**

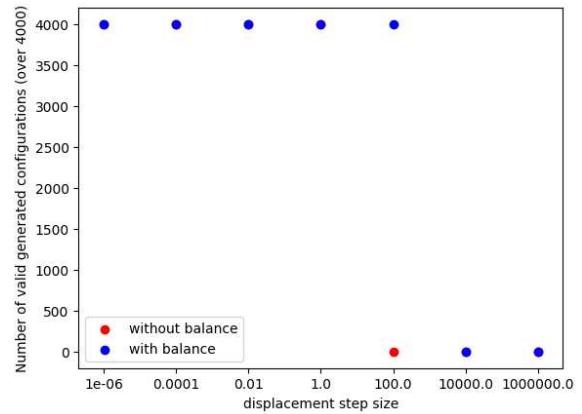
*Discussion:* Increasing the number of displacements require lower step sizes to reach the misclassification goal but it comes at the cost of more computations. However, increasing the number of displacements when the step size is already large results in incredibly large displacements, leading to invalid configurations in the MOTIV case.

#### RQ1.2: Are all generated adversarial configurations valid w.r.t. constraints in the VM?

As discussed in Section 3, we perform a basic type check on features. However, this check does not cover specific constraints such as cross-tree ones. To ensure the full compliance of our adversarial configurations, we run the analysis realised by the MOTIV video generator. This includes, amongst others, checking the correctness features values with respect to their specified intervals.

Figure 5 shows on the X-axis the different step sizes while the Y-axis depicts the number of valid adversarial configurations w.r.t. constraints. Regardless of the number of displacements and whether the training set is balanced, all results are the same except for Figure 5 that presents one difference for a displacement step size of 100. One possible explanation is that when the training set is balanced, more configurations can be taken as a starting point of the evasion algorithm: the gradient descent procedure might lead the current attack towards a slightly different area in which configurations remain valid. Overall, regardless of the number of authorized displacements, we can see a clear drop of valid configurations from 4,000 to 0 between step size set to 1 and 100.

*Discussion:* We can scope parameters such that adversarial configurations are *both successful and valid* when step size is set between 0.01 and 1.0, regardless of the number of displacements. Increasing the step size leads to non-valid configurations while with smaller step sizes, adversarial configurations have not moved enough to cross the separation of the classifier (leading to unsuccessful attacks).



**Figure 5: Number of valid attacks on class *acceptable*; X-axis represents different step size  $t$  values; Y-axis reports the number of valid configurations. In red and blue are respective results with an imbalance and a balance training set in terms of classes representation.**

**RQ1.3: Is using the evasion algorithm more effective than generating adversarial configurations with random displacements?** Previous results of RQ1.1 and RQ1.2 show we are able to craft valid adversarial configurations that can be misclassified by the ML classifier, but is our algorithm better than a random baseline? The baseline algorithm consists in: i) for each feature, choosing randomly whether to modify it; ii) choosing randomly to follow the slope of the gradient or going against it (the role of ‘-’ of line 5 in Algorithm 1 that can be changed into a ‘+’); iii) choosing randomly a degree of displacement (corresponding to the slope of the gradient ( $\nabla F(x^{m-1})$ ) of line 5 in Algorithm 1). Both the step size

and the number of displacements are the same as in the previous experiments.

Figure 6 shows the ability of random attacks to successfully mislead the classifier. Random modifications are not able to exceed 2,500 misclassifications (regardless of the number of displacements, the step size or whether the training set is balanced or not) which corresponds to more than half the generated configurations but with a lower effectiveness than with our evasion attack. The maximum number of misclassified configurations after random modifications starts from step size  $t = 10,000$  regardless of the studied number of displacements.

Considering the validity of these configurations, results are similar to what can be observed in Figure 5. The only difference is that the transition from 4,000 to 0 in the number of valid configurations is smoother and happens when  $t$  is in [0.01; 100].

*Discussion:* Previous results show that the effectiveness of evasion attacks are superior to random modifications since i) evasion attacks are able to craft configurations that are always misclassified by the ML classifier while less than 2,500 over 4,000 generations will be misclassified using random modifications; ii) generated evasion attacks support a larger set of parameter values for which generated configurations are valid; iii) we were able to identify sweet spots for which evasion attacks were able to generate 4,000 configurations that were both misclassified and valid.

#### RQ1.4: Are attacks effective regardless of the targeted class?

Previously, we generated evasion attacks from the class *non-acceptable* and tried to make them acceptable for the ML classifier but is our attack symmetric? Now, we configure our adversarial configuration generator so that it moves configurations from the class +1 (acceptable configurations) to the class -1 (non-acceptable).

Overall, the attack is symmetric: all generated adversarial configurations can be misclassified. Figure 7a shows that all generated configurations are misclassified when step size is set to 1 or higher with a number of displacements of 20 while, when the number of displacements is set to 100 (see Figure 7b), the step size can be set to 0.01 or higher. These observations are the same regardless of the balance in the training set.

Regarding the adversarial configuration validity, a transition from 4,000 to 0 can still be observed. However, when the number of displacements is set to 20 or when the training set is balanced, the transition is abrupt and occurs when step size belongs to the range [100, 10,000]. With a higher number of displacements (*i.e.*, 50 and 100 and no balance), the transition is smoother but happens with smaller step sizes (*i.e.*, with  $t$  in between [0.01; 100]. In the end, adversarial configurations can be generated regardless of the targeted class even if targeting the least represented class seems promising.

Our generated adversarial attacks are: 100% effective (always misclassified, RQ1.1), do not depend on the target class (RQ1.4) and yield valid configurations (RQ1.2). In contrast, our random baseline was only able to achieve 62.5% of effectiveness at best (RQ1.3). The balance in the training set does not affect these results and the targeted class affects show the same trends despite small differences (RQ1.4).

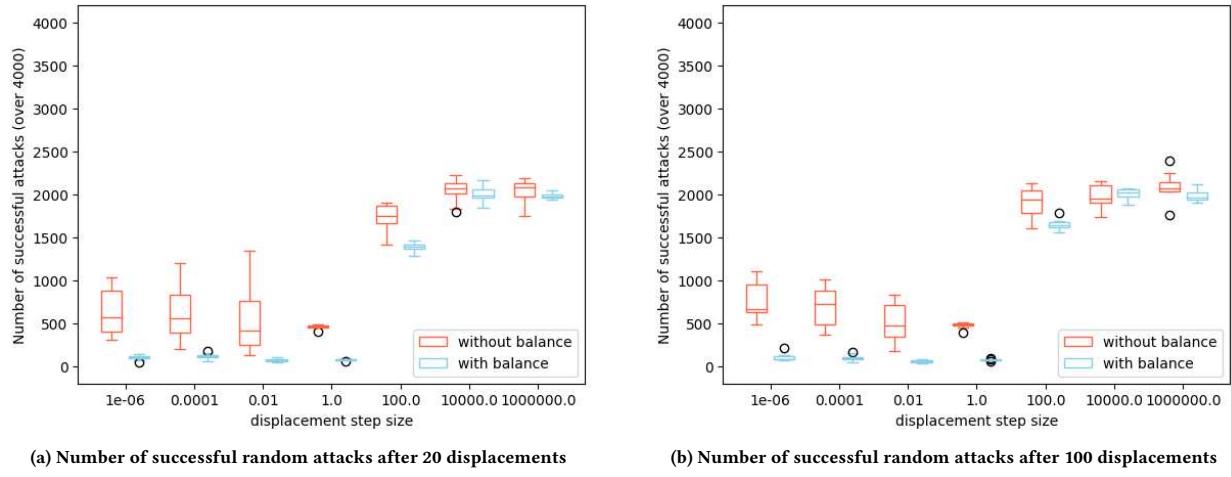
**4.3.2 RQ2: What is the impact of adding adversarial configurations to the training set regarding the performance of the classifier?** In our previous experiments, we only evaluated the impact of generated attacks in test sets. Yet, some ML techniques (GANs) take advantage of adversarial instances by incorporating them in the training set to improve the classifier confidence and possibly performance. In our context, we want to assess the impact of our attacks when our classifier includes them in the training dataset, especially with less “aggressive” (*e.g.*, small step sizes and a low number of displacements) configurations of the attacks.

To do so, we allowed 20 displacements in order to avoid configurations moving too far from their initial positions and we restrict the step size to every power of 10 in between  $10^{-4}$  and  $10^1$ . For each step size, we generate 25 adversarial configurations that are added all at once in the training set, we retrain the classifier and evaluate it on the configurations that constitute the initial test set (without any adversarial configurations in it). Every retraining process were repeated ten times in order to mitigate the effects of the random configuration selection and starting configurations. We also present results when the training set is balanced, in which case we have also augmented the test set to bring balance and to follow the same data distribution. In this case, the test set does not contain 4,000 configurations but about 7,000 in which 50% of the configurations are considered acceptable and the remaining are considered non-acceptable.

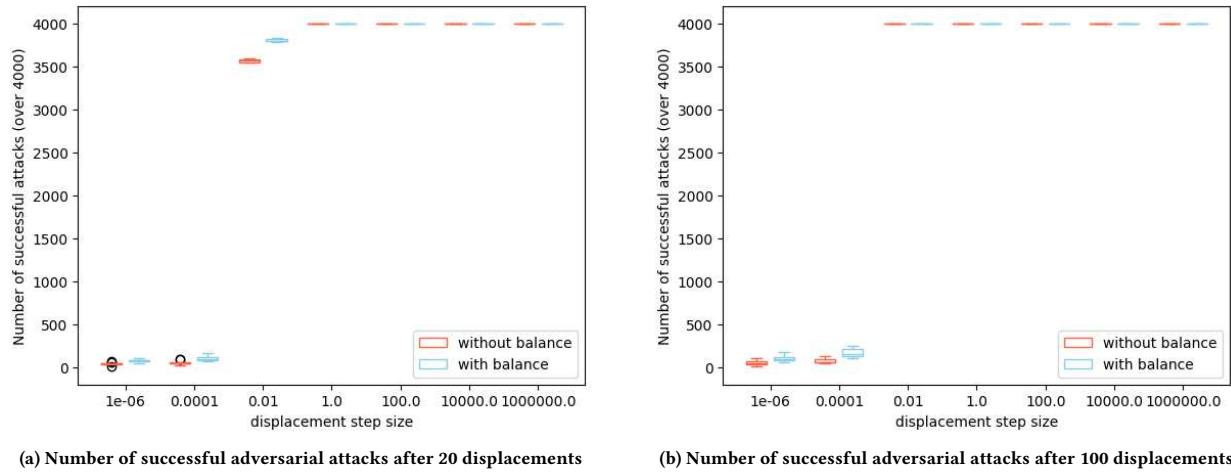
Figure 8 shows the accuracy of the retrained classifiers over a test set composed of 4,000 configurations for the red part and 7,000 configurations for the blue one.

The initial accuracy of the classifier was 96.4562% over the same 4,000 configurations and is shown as the horizontal red line. We make the following observations: i) using adversarial configurations in the training, even with low step sizes, tend to decrease the accuracy of the retrained classifier; ii) starting from step sizes of 1, every run gives the same result.

Specifically, with step size equals to  $10^{-4}$ , the median of the boxplot is very close to the initial accuracy (*i.e.*, 96.4562%) of the classifier and the interquartile range is small suggesting that the impact of adding adversarial configurations into the training set is marginal. Between  $10^{-3}$  to  $10^2$ , the median is slightly decreasing and the interquartile range tends to increase. At  $t = 10^{-1}$  the accuracy of the classifier drops to 91%, the adversarial configurations are specially efficient, forcing the ML classifier to change drastically its separation resulting in a lot of prediction errors. The last two step sizes shows that all the runs give the same results in terms of accuracy. Focusing on these runs, adversarial configurations had features with the same value: the amount of heat haze, of blur, of compression artefact or the amount of static noise of the 25 adversarial configurations are all equal. All of these features are directly related to the global quality of images, and are key for the classifier accuracy. We explain the evolution of the classifier’s accuracy as a combination of the contribution of the 8 most important features and the constraints of the VM. For low step sizes ( $t \in [10^{-4}, 10^{-2}]$ ), displacements are modest and therefore perturbations are very limited, though slightly observable. The sweet spot is at  $t = 10^{-1}$ : the resulting displacement is important enough to change feature values so that the associated configurations are moved effectively towards the separation and fool the classifier. We computed the



**Figure 6: Number of successful random attacks on class *acceptable*; X-axis represents different step size values  $t$  while Y-axis is the number of misclassified adversarial configurations by the classifier. In red and blue are respective results with an imbalance and a balance training set in terms of classes representation.**



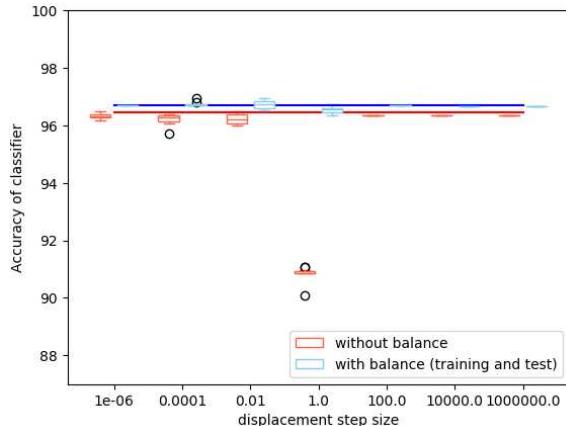
**Figure 7: Number of successful adversarial attacks on class *non-acceptable*; X-axis represents different step size values  $t$  while Y-axis is the number of misclassified adversarial configurations by the classifier; In orange and blue are respectively shown results when the training set is not balanced and when it is.**

means and standard deviations between the initial and adversarial configurations and their difference witnesses the impact of adversarial configurations on the classifier. For larger values of  $t$  (*i.e.*,  $>$  to  $10^{-1}$ ), these features lose their impact because their values are limited by constraints (so that they do not exceed the bounds).

In the case where training and test sets are balanced (in blue on Figure 8), results follow the same tendency. Since most of added configurations to provide balance are well classified, we see that the accuracy is a bit higher than in the non-balanced case. Values remain close to the baseline, however, when  $t = 1.0$ , results are worse than for other executions as for the non-balanced setting. Yet, we cannot conclude about the classifier robustness and more

experiments should be conducted to take into account the fact the balanced and non-balanced datasets do not contain the same number of configurations.

Our attacks cannot improve the classifier's accuracy but can make it significantly worse: 25 adversarial configurations over 500 can make the accuracy drops by 5%. Successful attacks also pinpoint visual features that do influence the videos' acceptability and that do make sense from the SPL perspective (computer vision).



**Figure 8: Accuracy of the classifier after retraining with 25 adversarial configurations in the training set over a test set of 4,000 configurations (7,000 configurations when the training set is balanced). In red are results when no balance are forced in the classes, in blue, both training set and test set are balanced. The initial accuracy of the classifier is represented by the horizontal line (96.4562% for the red line and 96.7143% for the blue one). X-axis represents different step size values  $t$  while Y-axis is the accuracy of the classifier (zoomed between 80% and 100%).**

## 5 THREATS TO VALIDITY

**Internal threats.** Choice of parameter values for our experiments may constitute a threat. The step size has been set to different powers of 10, we only used 3 different number of allowed displacements (*i.e.*, 20, 50 and 100). From our perspective, using step size of  $10^{-7}$  in a highly dimensional space seems ridiculously small while, on the contrary, using step size of  $10^4$  are tremendously large which motivates our choice to not going over these boundaries. However, the lower boundary could have been extended which might have affect results regarding *RQ2*. Still, given the design of our attack generator, it is likely that performance of the classifier would never have increased. Regarding the number of displacements, we could have used finer grained values. We sought a compromise between allowing a lot of small steps and a few big steps. Regarding the choice of evasion attacks, as presented in Section 2, several techniques exist. Evasion attacks showed interesting results and open new perspectives that we discuss in the Section 6.

We rely on centroids to deal with class imbalance (see Section 4.2). The centroid method has pros and cons: centroids are easy and quick to compute, new configurations tend to follow the same distribution as they result in more densely populated clusters and on rare occasions, make clusters expand a little bit. However, new configurations may not be realistic, since they do not provide so much diversity – centroids, by definition, lie in the middle of the cluster of points. Since our goal is only to limit imbalance in the available configurations, this technique is appropriate while maintaining the initial distribution of configurations. However, we are aware that other data augmentation techniques can be used.

**External threats.** We only assessed our adversarial attack generator on one case study, namely MOTIV. Yet, MOTIV is a complex and industrial case exhibiting various challenges for SPL analysis, including heterogeneous features, a large variability space and non-trivial non-functional aspects. The x.264 encoder has been studied (*e.g.*, [40]) but is relatively small in terms of features (only 16 were selected), heterogeneity (only Boolean features) and number of configurations. This can nevertheless be a candidate for replicating our study. Our adversarial approach is not specific to the video domain and, in principle, applicable to any SPL. Generating adversarial configurations without taking into account all constraints of the variability model directly into the attack algorithm may threaten the applicability of our approach to other SPLs. Calls to SAT/SMT solvers are unpractical due to feature heterogeneity and the frequency of validity checks. Benchmarks of large and real-world feature models can be considered if we are only interested in sampling aspects [34, 53]. Finally, open-source configurable systems like JHipster [31] can be of interest to study non-functional properties like binaries’ sizes or testing predictions. We also considered accuracy as a the main performance measure. Accuracy is the standard measure used in the advML literature [1, 7–9, 27, 41] to assess the impact of attacks.

## 6 DISCUSSIONS

Adversarial configurations pinpoint areas of the configuration space where the ML classifier fails or has low confidence in its prediction. We qualitatively discuss what *the existence of adversarial configurations suggests for an SPL* and to what extent the knowledge of adversarial configurations is actionable for MOTIV developers.

**#1 Adversarial training.** Firstly, developers might simply seek improvements of the ML classifier and making it more robust to attacks. Previous work on advML [1, 11, 22, 28, 37] proposed different defense strategies in presence of adversarial configurations. Adversarial training is a specific category of defense: the training sample is augmented with adversarial examples to make ML models more robust. In our case study, it consists in applying our attack generator and re-inject adversarial configurations as part of the original training set. We saw in *RQ2* that, when adversarial configurations are introduced in the training set, even moderately aggressive attacks affect the ML classifier performance. Our adversarial training is not adequate: our adversarial generator has simply not been designed for this defensive task and rather excels in finding misclassifications. It opens two perspectives. The first is to apply other, more effective defense mechanisms (manifold projections, stochasticity, prepossessing, *etc.* [1, 11, 22, 28, 37]). The second and most important perspective is to leverage adversarial ML knowledge for improving the SPL itself with “friendly” rather than malicious attacks, fooling the classifier is a mean to this objective.

**#2 Improvement of the testing oracle.** The labelling of videos as acceptable or non-acceptable – the testing oracle – is approximated by the ML classifier. If the oracle is not precise enough, it is likely that the approximation performs even worse. In the MOTIV case, oracles are an approximation of the human perception system which in turn could be seen as an approximation of the real separation between acceptable images and non-acceptable ones regarding a specific task. Object recognition should potentially work on an

infinite number of input images which makes the construction of a “traditional” oracle (a function that is able to give the nature of every single input) challenging. Testing oracles for an SPL are programs that may fail on some specific configurations. Adversarial configurations can lead to “cases” (videos) for which the oracle has not been designed and tested for and may provide insights to improve such oracles.

MOTIV’s developers may revise the visual assessment procedure to determine what a video of *sufficient quality means* [23, 56]. Adversarial configurations can help understanding the bugs (if any) of the procedure over specific videos (see Figure 3, page 5). Based on this knowledge, a first option is to fix this procedure – adversarial configurations would then act as good test cases for ensuring non-regression issues with the oracle. In our context, one can envision to crowd-source the labelling effort with humans (e.g., with Amazon Mechanical Turk [15]). However, asking human beings to check whether a video is acceptable or not is costly and hardly scalable – we have derived more than 4,000 videos. Crowd-sourcing is also prone to errors made by humans due to fatigue or disagreements on the task. To decrease the effort, adversarial configurations can be specifically reviewed as part of the labelling process. An open problem is to find a way to control adversarial displacements such that we are able to ensure that the generated adversarial configuration does not cross the ML separation. This level of control is left for future work. Overall, the choice of the adequate testing oracle strategy in the MOTIV case is beyond the scope of this paper. Several factors are involved, including cost (e.g., manually labelling videos has a significant cost) and reliability.

**#3 Improvement of the variability model.** While generating adversarial configurations, SPL practitioners can gain insights on whether the feature model is under or over constrained. Looking at modified features of adversarial configurations (see RQ2), practitioners can observe that the same patterns arise involving some features or combinations of features. Such behavior typically indicate that constraints are missing – some configurations are allowed despite they should not be but it was never specifically defined as such in the variability model. Conversely, adversarial configurations can also help identifying which constraints can be relaxed.

**#4 Improvement of the variability implementation.** Features of MOTIV are implemented in Lua [32]. An incorrect implementation can be the cause of non-acceptable configurations either because of bugs in individual features or undesired feature interactions. In the case of MOTIV, we did not find variability-related bugs. We rather considered that the cause of non-acceptable videos was due to the variability model and that the solution was to add constraints preventing this.

## 7 RELATED WORK

Our contribution is at the crossroad of (adversarial) ML, constraint mining, variability modeling, and testing.

**Testing and learning SPLs.** Testing all configurations of an SPL is most of time challenging and sometimes impossible, due to the exponential number of configurations [30, 36, 38, 46, 59–61]. ML techniques have been developed to reduce cost, time and energy of deriving and testing new configurations using inference

mechanisms. For instance, regression models can be used to perform performance prediction of configurations that have not been generated yet [29, 42, 45, 48, 51, 52, 58]. In [55, 56], we proposed to use supervised ML to discover and retrieve constraints that were not originally expressed before in a variability model. We used decision trees to create a boundary between the configurations that should be discarded and the ones that are allowed. In this paper, we build upon previous works and follow a new research direction with SVM-based adversarial learning.

Siegmund et al. [53] reviewed ML approaches on variability models. They propose THOR, a tool for synthesizing realistic attributed variability models. An important issue in this line of research is to assess the robustness of ML on variability models. Yet, our work specifically aims to improve ML classifiers of SPL. None of these bodies of work use adversarial ML neither the possible impact that adversarial configurations could have on the predictions.

**Adversarial ML** can be seen as set of security assessment and reinforcement techniques helping to better understand flaws and weaknesses of ML algorithms. Typical scenarios in which adversarial learning is used are: network traffic monitoring, spam filtering, malware detection [1, 6–10] and more recently autonomous cars and object recognition [24, 25, 35, 43, 44, 50, 62]. In such works, authors suppose that a system uses ML in order to perform a classification task (e.g., differentiate emails as spams and non-spams) and some malicious people try to fool such classification system. These attackers can have knowledge on the system such as the dataset used, the kind of ML technique that is used, the description of data, etc. The attack then consists in crafting a data point in the description space that the ML algorithm will misclassify. Recent works [27] used adversarial techniques to strengthen the classifier by specifically creating data that would induce such kind of misclassification. In this paper, we propose to use a similar approach but adapted to SPL engineering: adversarial techniques may be used to strengthen the SPL (including variability model, implementation and testing oracle over products) while analyzing a small set of configurations. To our knowledge, no adversarial technique has been experimented in this context.

## 8 CONCLUSION

Machine learning techniques are increasingly used in software product line engineering as they are able to predict whether a configuration (and its associated program variant) meets quality requirements. ML techniques can make prediction errors in areas where the confidence in the classification is low. We adapted adversarial techniques on our MOTIV case and generated both successful and valid attacks that can fool a classifier with a low number of adversarial configurations and decrease its performance by 5%. The analysis of the attacks exhibit the influence of important features and variability model constraints. This is a first and promising step in the direction of using adversarial techniques as a novel framework for quality assurance of software product lines. As future work, we plan to compare adversarial learning with traditional learning or sampling techniques (e.g., random, t-wise). Generally we want to use adversarial ML to support quality assurance of SPLs.

## REFERENCES

- [1] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D Joseph, and J Doug Tygar. 2006. Can machine learning be secure?. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*. ACM, New York, NY, USA, 16–25.
- [2] Don S. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *SPLC'05 (LNCS)*, Vol. 3714. Springer, Berlin, Germany, 7–20.
- [3] Richard Bellman. 1957. *Dynamic Programming* (1 ed.). Princeton University Press, Princeton, NJ, USA.
- [4] David Benavides, Sergio Segura, and Antonio Ruiz-Cortes. 2010. Automated Analysis of Feature Models 20 years Later: a Literature Review. *Information Systems* 35, 6 (2010), 615–636.
- [5] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '13)*. ACM, New York, NY, USA, Article 7, 8 pages. <https://doi.org/10.1145/2430502.2430513>
- [6] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. 2013. Evasion attacks against machine learning at test time. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer Berlin, Berlin, Heidelberg, 387–402.
- [7] B. Biggio, L. Didaci, G. Fumera, and F. Roli. 2013. Poisoning attacks to compromise face templates. In *2013 International Conference on Biometrics (ICB)*. IEEE, New York, USA, 1–7. <https://doi.org/10.1109/ICB.2013.6613006>
- [8] Battista Biggio, Giorgio Fumera, and Fabio Roli. 2014. Pattern recognition systems under attack: Design issues and research challenges. *International Journal of Pattern Recognition and Artificial Intelligence* 28, 07 (2014), 1460002.
- [9] Battista Biggio, Giorgio Fumera, and Fabio Roli. 2014. Security evaluation of pattern classifiers under attack. *IEEE transactions on knowledge and data engineering* 26, 4 (2014), 984–996.
- [10] Battista Biggio, Blaine Nelson, and Pavel Laskov. 2012. Poisoning Attacks Against Support Vector Machines. In *Proceedings of the 29th International Conference on International Conference on Machine Learning (ICML '12)*. Omnipress, USA, 1467–1474. <http://dl.acm.org/citation.cfm?id=3042573.3042761>
- [11] Battista Biggio and Fabio Roli. 2018. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition* 84 (2018), 317–331.
- [12] Eric Bodden, Társis Tolédo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPLIFT : statically analyzing software product lines in minutes instead of years. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16–19, 2013*. ACM, New York, USA, 355–364. <https://doi.org/10.1145/2491956.2491976>
- [13] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. 2010. Introducing TVL, a Text-based Feature Modelling. In *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27–29, 2010. Proceedings (ICB-Research Report)*, David Benavides, Don S. Batory, and Paul Grünbacher (Eds.), Vol. 37. Universität Duisburg-Essen, Essen, Germany, 159–162. [http://www.vamos-workshop.net/proceedings/VaMoS\\_2010\\_Proceedings.pdf](http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf)
- [14] Tom Brown, Dandelion Mane, Aurko Roy, Martin Abadi, and Justin Gilmer. 2017. Adversarial Patch. <https://arxiv.org/pdf/1712.09665.pdf> (2017).
- [15] Michael Buhrmester, Tracy Kwang, and Samuel D Gosling. 2011. Amazon's Mechanical Turk: A new source of inexpensive, yet high-quality, data? *Perspectives on psychological science* 6, 1 (2011), 3–5.
- [16] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.
- [17] Andreas Classen, Quentin Boucher, and Patrick Heymans. 2011. A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL. *Science of Computer Programming, Special Issue on Software Evolution, Adaptability and Variability* 76, 12 (2011), 1130–1143.
- [18] Paul Clements and Linda M. Northrop. 2001. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, Boston, USA.
- [19] Jean-Marc Davril, Patrick Heymans, Guillaume Bécan, and Mathieu Acher. 2015. On Breaking The Curse of Dimensionality in Reverse Engineering Feature Models. In *17th International Configuration Workshop (17th International Configuration Workshop)*, Vol. 17th International Configuration Workshop. Vienna, Austria. <https://hal.inria.fr/hal-01243571>
- [20] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. 2018. On the Intriguing Connections of Regularization, Input Gradients and Transferability of Evasion and Poisoning Attacks. *CoRR* abs/1809.02861 (2018). arXiv:1809.02861 <http://arxiv.org/abs/1809.02861>
- [21] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. 2019. Why Do Adversarial Attacks Transfer? Explaining Transferability of Evasion and Poisoning Attacks. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA. <https://www.usenix.org/conference/usenixsecurity19/>
- [22] Guneet S Dhillon, Kamyr Azizzadenesheli, Zachary C Lipton, Jeremy Bernstein, Jean Kossaifi, Aran Khanna, and Anima Anandkumar. 2018. Stochastic activation pruning for robust adversarial defense. *arXiv preprint arXiv:1803.01442* (2018).
- [23] Richard W. Doselman and Xue Dong Yang. 2012. *No-Reference Noise and Blur Detection via the Fourier Transform*. Technical Report. University of Regina, CANADA.
- [24] Gamaleldin F Elsayed, Shreya Shankar, Brian Cheung, Nicolas Papernot, Alex Kurakin, Ian Goodfellow, and Jascha Sohl-Dickstein. 2018. Adversarial Examples that Fool both Human and Computer Vision. *arXiv preprint arXiv:1802.08195* (2018).
- [25] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. 2017. Robust Physical-World Attacks on Deep Learning Models. *arXiv preprint arXiv:1707.08945* 1 (2017).
- [26] José Angel Galindo Duarte, Mauricio Alférrez, Mathieu Acher, Benoît Baudry, and David Benavides. 2014. A Variability-Based Testing Approach for Synthesizing Video Sequences. In *ISSTA '14: International Symposium on Software Testing and Analysis*. San José, California, United States. <https://hal.inria.fr/hal-01003148>
- [27] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems*. 2672–2680.
- [28] Chuan Guo, Mayank Rana, Moustapha Cisse, and Laurens van der Maaten. 2017. Countering adversarial images using input transformations. *arXiv preprint arXiv:1711.00117* (2017).
- [29] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *ASE*.
- [30] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoît Baudry. 2018. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empirical Software Engineering* (July 2018). <https://doi.org/10.07980 Empirical Software Engineering journal>
- [31] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoît Baudry. 2019. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empirical Software Engineering* 24, 2 (2019), 674–717. <https://doi.org/10.1007/s10664-018-9635-4>
- [32] Roberto Ierusalimschy. 2006. *Programming in Lua, Second Edition*. Lua.Org.
- [33] Cem Kaner, James Bach, and Bret Pettichord. 2001. *Lessons Learned in Software Testing*. John Wiley & Sons, Inc., New York, NY, USA.
- [34] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2018. Is There a Mismatch between Real-World Feature Models and Product-Line Research?. In *Software Engineering und Software Management 2018, Fachtagung des GI-Fachbereichs Softwaretechnik, SE 2018, 5.–9. März 2018, Ulm, Germany. (LNI)*, Matthias Tichy, Eric Bodden, Marco Kuhrmann, Stefan Wagner, and Jan-Philipp Steghöfer (Eds.), Vol. P-279. Gesellschaft für Informatik, 53–54. <https://dl.gi.de/20.500.12116/16312>
- [35] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2016. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533* (2016).
- [36] Axel Legay and Gilles Perrouin. 2017. On Quantitative Requirements for Product Lines. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '17)*. ACM, New York, NY, USA, 2–4. <https://doi.org/10.1145/3023956.3023970>
- [37] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2017. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083* (2017).
- [38] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 643–654. <https://doi.org/10.1145/2884781.2884793>
- [39] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining configuration constraints: static analyses and empirical results. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - Jun 07, 2014*, 140–151. <https://doi.org/10.1145/2568225.2568283>
- [40] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using bad learners to find good configurations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4–8, 2017*. Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 257–267. <https://doi.org/10.1145/3106237.3106238>
- [41] Blaine Nelson, Marco Barreno, Fuching Jack Chi, Anthony D Joseph, Benjamin IP Rubinstein, Udayan Saini, Charles Sutton, J Doug Tygar, and Kai Xia. 2008. Exploiting Machine Learning to Subvert Your Spam Filter. *LEET* 8 (2008), 1–9.
- [42] Jeho Oh, Don S. Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4–8, 2017*, 61–71. <https://doi.org/10.1145/3106237.3106273>
- [43] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. 2016. The Limitations of Deep Learning in Adversarial Settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*. 372–387. <https://doi.org/10.1109/>

- EuroSP.2016.36  
[44] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 1–18. <https://doi.org/10.1145/3132747.3132785>
- [45] Juliana Alves Pereira, Hugo Martin, Mathieu Acher, Jean-Marc Jälzälquel, Goetz Botterweck, and Anthony Ventresque. 2019. Learning Software Configuration Spaces: A Systematic Literature Review. arXiv:arXiv:1906.03018
- [46] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22–27, 2019*. 240–251. <https://doi.org/10.1109/ICST2019.00032>
- [47] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag.
- [48] A. Sarkar, Jianmei Guo, N. Siegmund, S. Apel, and K. Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T). In *ASE'15*.
- [49] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic semantics of feature diagrams. *Comput. Netw.* 51, 2 (2007), 456–479.
- [50] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K Reiter. 2016. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1528–1540.
- [51] Norbert Siegmund, Alexander Grebhahn, Christian Kästner, and Sven Apel. [n.d.]. Performance-Influence Models for Highly Configurable Systems. In *ESEC/FSE'15*.
- [52] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. 2013. Scalable Prediction of Non-functional Properties in Software Product Lines: Footprint and Memory Consumption. *Inf. Softw. Technol.* (2013).
- [53] Norbert Siegmund, Stefan Sobernig, and Sven Apel. 2017. Attributed Variability Models: Outside the Comfort Zone. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 268–278. <https://doi.org/10.1145/3106237.3106251>
- [54] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. 2018. Variability-based model transformation: formal foundation and application. *Formal Asp. Comput.* 30, 1 (2018), 133–162. <https://doi.org/10.1007/s00165-017-0441-3>
- [55] Paul Temple, Mathieu Acher, Jean-Marc Jézéquel, and Olivier Barais. 2017. Learning Contextual-Variability Models. *IEEE Software* 34, 6 (2017), 64–70. <https://doi.org/10.1109/MS.2017.4121211>
- [56] Paul Temple, José Angel Galindo Duarte, Mathieu Acher, and Jean-Marc Jézéquel. 2016. Using Machine Learning to Infer Constraints for Product Lines. In *Software Product Line Conference (SPLC)*. Beijing, China. <https://doi.org/10.1145/2934466.2934472>
- [57] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. 2016. Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *J. Log. Algebr. Meth. Program.* 85, 2 (2016), 287–315. <https://doi.org/10.1016/j.jlamp.2015.11.006>
- [58] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Laura Semini. 2016. Variability-Based Design of Services for Smart Transportation Systems. In *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, IsoLA 2016, Imperial, Corfu, Greece, October 10–14, 2016, Proceedings, Part II*. 465–481. [https://doi.org/10.1007/978-3-319-47169-3\\_38](https://doi.org/10.1007/978-3-319-47169-3_38)
- [59] Maurice H. ter Beek and Axel Legay. 2019. Quantitative Variability Modeling and Analysis. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS '19)*. ACM, New York, NY, USA, Article 13, 2 pages. <https://doi.org/10.1145/3302333.3302349>
- [60] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* (2014).
- [61] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A classification of product sampling for software product lines. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10–14, 2018*. 1–13. <https://doi.org/10.1145/3233027.3233035>
- [62] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 132–142. <https://doi.org/10.1145/3238147.3238187>

# Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features

Daniel-Jesús Muñoz

CAOSD, Dpt. LCC, Universidad de Málaga, Andalucía Tech  
Málaga, Andalucía, Spain  
danimg@lcc.uma.es

Mónica Pinto, Lidia Fuentes

CAOSD, Dpt. LCC, Universidad de Málaga, Andalucía Tech  
Málaga, Andalucía, Spain  
{pinto,lff}@lcc.uma.es

## ABSTRACT

Analyses of *Software Product Lines (SPLs)* rely on automated solvers to navigate complex dependencies among features and find legal configurations. Often these analyses do not support numerical features with constraints because propositional formulas use only Boolean variables. Some automated solvers can represent numerical features natively, but are limited in their ability to count and *Uniform Random Sample (URS)* configurations, which are key operations to derive unbiased statistics on configuration spaces.

*Bit-blasting* is a technique to encode numerical constraints as propositional formulas. We use bit-blasting to encode Boolean and numerical constraints so that we can exploit existing #SAT solvers to count and URS configurations. Compared to state-of-art Satisfiability Modulo Theory and Constraint Programming solvers, our approach has two advantages: 1) faster and more scalable configuration counting and 2) reliable URS of SPL configurations. We also show that our work can be used to extend prior SAT-based SPL analyses to support numerical features and constraints.

## CCS CONCEPTS

- Theory of computation → Logic and verification; Automated reasoning;
- Software and its engineering → Software product lines; Model-driven software engineering; Software performance.

## KEYWORDS

feature model, bit-blasting, propositional formula, numerical features, model counting, software product lines

### ACM Reference Format:

Daniel-Jesús Muñoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336297>

Jeho Oh

Department of Computer Science  
Austin, Texas, USA  
jeho@cs.utexas.edu

Don Batory

Department of Computer Science  
Austin, Texas, USA  
batory@cs.utexas.edu

France. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3336294.3336297>

## 1 INTRODUCTION

*Software Product Lines (SPLs)* are highly configurable systems. A *feature model* defines the variability of an SPL using features and constraints. A *feature* is an increment in program functionality. A constraint is a relationship among features, where the presence or absence of some features requires or precludes other features. A valid combination of features is a *configuration*. All configurations define a *configuration space* [2].

Classical feature models use *Boolean features* that have only two values (present, absent). Boolean features are insufficient for real-world SPLs, as there exist features that have a series or a range of numbers as explicit values. An example is the size in bytes of a datafile [49]; it is represented by a power of 10 series of values in a feature model. These features are called *Numerical Features (NFs)*. Feature models with NFs are *Numerical Feature Models (NFMFs)*.

It is infeasible to understand large configuration spaces by enumeration. Most SPLs do not have an analytical model to accurately predict run-time properties (eg., [48]), so it is common to sample configurations, build the product for each sample, and gather data about samples by benchmarking. Doing so creates a dataset on the configuration space. This approach has been used many times: deriving the influence of a feature for performance modeling [33, 55], performing multi-objective optimization [26, 34, 35, 56], and evaluating different sampling approaches to locate variability bugs [47, 62].

Counting the number of configurations and *Uniform Random Sampling (URS)* configurations are two operations for unbiased statistical inferences on SPLs. Counting and URS solutions of NFMFs, however, are largely unexplored. Only a handful of automated solvers can represent and reason over both Boolean and numerical feature constraints, namely *Satisfiability Modulo Theories (SMT)* [6] and *Constraint Programming (CP)* [53] solvers. Unfortunately, SMT and CP solvers cannot count the number of configurations (except by enumeration) or uniform sample configurations. Prior work sampled configurations with SMT and CP solvers, but whether the produced samples are uniformly distributed was *not* shown.

In contrast, for classical feature models, there are tools that can count faster than SMT and CP solvers and enable URS of configurations. Every classical feature model can be encoded as a propositional formula, where a solution of the propositional formula is a valid configuration of the feature model. #SAT solvers extend *Satisfiability (SAT)* solvers to count the number of solutions of a propositional formula without enumeration [12]. Chakraborty et al. and Oh et al. developed tools to URS solutions of a propositional formula, based on #SAT technology [17, 51].

*Bit-blasting* encodes numerical values as binary bits and represent operations on them as propositional formulas [15]. We propose to represent NFM s and their constraints by bit-blasting and utilize existing SAT-based tools for counting and URS classical feature models. We make use of the ‘Tactic’ functionality of the Z3 SMT solver [22] to convert NFM s and their constraints into propositional formulas using bit-blasting, which are then integrated with the propositional formulas of classical feature models. This allows us to represent NFM s as a *Bit-Blasted Propositional Formula (BBPF)*.

BBPF can be input to existing #SAT-based tools for counting and URS solutions of a NFM, which SMT and CP solvers cannot do. In this way NFM s can be analyzed by existing tools with minimal extra work. The contributions of our work are:

- Use of bit-blasting to express NFM s and constraints,
- Integration of bit-blasting and classical feature models to translate a NFM into a BBPF,
- Experiments that show counting and URS solutions of BBPF outperform SMT and CP solvers, and
- Evaluation of known SPL analyses using NFM s with huge configuration spaces, the largest exceeding  $10^{45}$  products.

## 2 BACKGROUND

### 2.1 Bit-Blasting

*Bit-blasting* or *flattening* is the transformation of a bit-vector arithmetic formula to an equivalent propositional formula [3]. It has been mainly used in hardware verification [19] and to optimize the hardware verification task itself [27, 66]. Brillout et al. [13] used bit-blasting to create a bit-accurate and complete decision procedure for IEEE-compliant binary floating-point arithmetic units.

We focus on the following arithmetic operations: equality ( $=$ ), inequalities ( $\neq, >, \geq$ ), addition ( $+$ ) and subtraction ( $-$ ). Although bit-blasting supports more operations, it is known that multiplication and division do not scale with increasing bit-width [15]. Real-world SPLs that we have studied are described in Table 2. They largely limit their use of numerical operations to equality and inequalities. A few add two NFM s and compare the result to a constant. Technical details on bit-blasting are covered later in Section 3.

### 2.2 Feature Models

A classical feature model uses only Boolean features but this very restriction allows it to be transformed into a propositional formula, where features are variables and constraints are clauses [2]. Many tools can convert an feature model into a propositional formula. One is FeatureIDE that exports a feature model written in their tool as a *Conjunctive Normal Form (CNF)* formula [63]. Another is KClause which transforms a KConfig model into a compact CNF formula [38].

Real-world SPLs use NFM s that contain both binary features and NFM s [36]. An NF has a name  $N$ , a type (ie., domain), and range (eg.,  $N \in [1, 2, \dots, 128]$ ). NFM s add new constraints to the set of propositional connectives, including: numerical equality ( $=$ ), numerical inequalities ( $\neq, >, <, \geq$  and  $\leq$ ), and occasionally addition and subtraction but no other numerical operations (at least in KConfig systems [28, 29]). NFM s can also have constraints with Boolean features, where the value of an NF affects the value of a Boolean feature, and vice versa.

Two examples of NFM s are: (1) the HADAS eco-assistant [48] where energy context parameters are represented as NFM s in an Integer domain, and propositional connectives and inequalities are present in cross-tree constraints (eg.,  $AES\_crypto \Rightarrow key\_size > 128$ ) and (2) WeaFQAs [37] where some variables of quality attributes are NFM s with Integer or Float domains, containing propositional connectives and interval constraints (ie., numerical value ranges).

### 2.3 Uniform Random Sampling and Finding Sub-Optimal Products in Colossal Spaces

Uniform sampling ensures all samples are valid and uniformly distributed across the configuration space, so that the samples can be used for standard statistic approaches.

Oh et al. [51] were the first to URS an SPL configuration space. They used the following ideas: Let  $\phi$  be the propositional formula of a classical feature model. Let  $S(\phi)$  be the set of all solutions of  $\phi$ . Each solution of  $\phi$  is in a 1-to-1 correspondence with a configuration product in the feature model [2].

Let  $|S(\phi)|$  be the number of solutions in  $S(\phi)$ . A uniform random number generator can select an integer  $j$  in the range  $[1..|S(\phi)|]$ . The trick is to convert  $j$  into the  $j^{th}$  configuration in a fixed linear ordering of  $S(\phi)$ . By construction, URS of numbers in  $[1..|S(\phi)|]$  is isomorphic to URS of configurations in  $S(\phi)$ .

SAT solvers find solutions to a given  $\phi$ . #SAT solvers, a relatively new SAT technology, can count  $|S(\phi)|$ . And #SAT can also be used to convert an integer  $j$  into the  $j^{th}$  configuration of  $S(\phi)$  [38]. Here’s how: Let  $F = (f_1, f_2, \dots)$  be a fixed-order list of all features in a feature model. A #SAT tool can count  $n = |S(\phi \wedge f_1)|$ , the number of solutions that have feature  $f_1$ . If  $j \leq n$ , then the  $j^{th}$  configuration must have feature  $f_1$  otherwise it has  $\neg f_1$ . Repeating this logic on the remaining features in  $F$  performs a binary search on  $S(\phi)$  to reveal the presence or absence of every feature in the  $j^{th}$  configuration.

Here is a great application for URS: it can be used to quickly locate sub-optimal products in  $S(\phi)$ . Take  $n$  URS in  $S(\phi)$ , build and benchmark each of them. Let  $c_{best}$  be the best performing configuration among these  $n$  samples. Oh et al. [51] showed that  $c_{best}$  will be, on average, within the top  $\frac{1}{n+1}$  percentile of the best performing configurations in  $S(\phi)$ . So if 99 uniformly random samples are taken,  $c_{best}$  is in the top 1% of the best performing configurations of  $S(\phi)$ , on average, no matter how big  $|S(\phi)|$  is [51]. We explore this application further on Section 4.

## 3 BIT-BLASTING FOR NFM S

We describe how to integrate bit-blasting and classical feature models to form NFM s and how to translate a NFM into a BBPF.

#	Operation	Bit-Blasted Model	Propositional Formula
1	$(NF_a == NF_b)$	$(a_3 == b_3) \wedge (a_2 == b_2) \wedge (a_1 == b_1)$	$(a_3 \Leftrightarrow b_3) \wedge (a_2 \Leftrightarrow b_2) \wedge (a_1 \Leftrightarrow b_1)$
2	$(NF_a \neq NF_b)$	$(a_3 \neq b_3) \vee (a_2 \neq b_2) \vee (a_1 \neq b_1)$	$(a_3 \oplus b_3) \vee (a_2 \oplus b_2) \vee (a_1 \oplus b_1)$
3	$(NF_a > NF_b)$	$(a_3 < b_3) \vee ((a_3 == b_3) \wedge (a_2 > b_2)) \vee ((a_3 == b_3) \wedge (a_2 == b_2) \wedge (a_1 > b_1))$	$(\neg a_3 \wedge b_3) \vee ((a_3 \Leftrightarrow b_3) \wedge (a_2 \wedge \neg b_2)) \vee ((a_3 \Leftrightarrow b_3) \wedge (a_2 \Leftrightarrow b_2) \wedge (a_1 \wedge \neg b_1))$
4	$(NF_a \geq NF_b)$	$(a_3 < b_3) \vee ((a_3 == b_3) \wedge (a_2 \geq b_2)) \vee ((a_3 == b_3) \wedge (a_2 == b_2) \wedge (a_1 \geq b_1))$	$(\neg a_3 \wedge b_3) \vee ((a_3 \Leftrightarrow b_3) \wedge (b_2 \Rightarrow a_2)) \vee ((a_3 \Leftrightarrow b_3) \wedge (a_2 \Leftrightarrow b_2) \wedge (b_1 \Rightarrow a_1))$
5	$(NF_a \pm NF_b)$	$S_1^4 \equiv [(a_1 \oplus b_1) \oplus C_0, (a_2 \oplus b_2) \oplus C_1, (a_3 \oplus b_3) \oplus C_2, C_3]$ $C_1^3 \equiv (a_i \wedge b_i) \vee C_{i-1}$ $C_0 \equiv ('+' \Rightarrow 0) \wedge ('-' \Rightarrow 1)$	$[(a_1 \oplus b_1) \oplus \pm, (a_2 \oplus b_2) \oplus ((a_1 \wedge b_1) \vee \pm), (a_3 \oplus b_3) \oplus ((a_2 \wedge b_2) \vee ((a_1 \wedge b_1) \vee \pm)), (a_3 \wedge b_3) \vee ((a_2 \wedge b_2) \vee ((a_1 \wedge b_1) \vee \pm))]$

Table 1: Bit-blasted Models and propositional formula Transformation Examples for 2-bit two's Complement Signed Integers

### 3.1 Bit-Blasting for Arithmetic Operations

This section reviews ideas about bit-blasting that are known to be implemented by Z3. Bit-vectors have two properties: width of the vector and whether it is unsigned (binary *sign-magnitude* encoding) or signed (binary *two's complement*<sup>1</sup> encoding). We use the Little-Endian representation, *i.e.*, signed bit-vectors, where the last bit encodes the sign as positive (0) or negative (1).<sup>2</sup>

Table 1 shows examples of two's compliment bit-blasting propositional formulas for equality, inequality, greater, greater or equal, and addition/subtraction of Little-Endian signed integers with a value range of [-4, 3] (*i.e.*,  $n = 3$  bits) where  $a_3$  is the integer sign. Of course, a greater number of bits can be used in Table 1, but  $n=3$  shows the repeating patterns in propositional formula that bit-blasting uses. Equality ( $==$ ) is the conjunction of bit-by-bit equivalences (row 1, col propositional formula). Inequality ( $\neq$ ) is a bit-by-bit disjunction of logical *XORs* ( $\oplus$ ) (row 2, col propositional formula). After the numerical sign comparison (first clause of col PF in rows 3 and 4), there are bit-by-bit equivalences till the last bit of the series, which involve an implication in case of  $\geq$  (row 4, col 3), or a disjunction of opposites in case of  $>$  (row 3, col 3).

Bit-blasting addition is harder. Addition of bit-vectors can create a result outside the range of the operands due to the number of bits necessary to represent the result. For example, for 3 signed bits, if we perform '3 + 1', the result is '4', which is impossible to represent with 3 signed bits; we need 4 signed bits. The extra bit is called a *carry bit*. Then, a binary addition requires two data inputs, and produces two outputs, the Sum (S) of the equation and a Carry (C) bit as shown in the operation 5 of Table 1. Subtraction in a two's complement encoding is an addition differing on  $C_0$ , which is 0 for addition operations, and 1 for subtraction operations.

SAT solvers regularly work with propositional formulas in CNF form [12]. To transform the propositional formulas of Table 1, the Z3 solver uses Tseitin's CNF transformations with skolemization [65], as it is a widely known method to transform propositional formulas into a CNF formula while maintaining the model satisfiability and number of configurations.

### 3.2 Producing a $\text{BBPF}$ for an $\text{NFM}$

We encode the Boolean features and their constraints of an  $\text{NFM}$  as a propositional formula in the standard way [2]. Then,  $\text{NFs}$  and their constraints of a  $\text{NFM}$  are encoded as propositional clauses making use of the Z3 solver 'Tactic' functionality.<sup>3</sup> We conjoin both predicates (or substitute them below) to form the  $\text{BBPF}$  for that  $\text{NFM}$ . Here are some details:

**NF Definition.** Let a signed  $\text{NF } f$  have range  $[a, b]$ . Bit-blasting uses  $\lceil \log_2(\max(|a|, |b|)) + 1 \rceil + 1$  variables to represent the bits of  $f$ , where 1 variable encodes the sign. Propositional clauses for two constraints ( $f \geq a$ ) and ( $f \leq b$ ) are conjoined to limit the range of  $f$  values. If applicable, the range of  $f$  is shifted to  $[0, b - a]$  as it may simplify the formula and use fewer bits, namely  $\lceil \log_2(b - a) + 1 \rceil + 1$ .

We represent all  $\text{NFs}$  as integers. Decimal point values can be represented by shifting the points to the desired precision, which is shifted back when the configurations are sampled.

**NF Constraints.** Constraints between  $\text{NFs}$  can be directly derived as propositional clauses from bit-blasting and conjoined to a propositional formula. If an  $\text{NF}$  is a constant, its binary value is used, which can simplify the formula by Boolean constraint propagation.

Two  $\text{NFs}$  bounded under the same constraint may have different bit-widths due to different value ranges. As the bit-width of each  $\text{NF}$  is fixed, the  $\text{NF}$  with shorter bit-width needs to be extended to

<sup>1</sup>Two's complement negative integer encoding is the binary complement of the positive encoding plus one bit.

<sup>2</sup>Little-Endian: An order of bits in which the "little end" (least significant value in the sequence) is represented first in the sequence.

<sup>3</sup>We have configured Z3 solver to convert  $\text{NFs}$  and constraints into propositional formulas with the 'Tactic' functionality command <Then('simplify', 'bit-blast', 'tseitin-cnf')> However, Z3 solver is not primarily intended to be used for this task, nor it is a well-documented functionality.

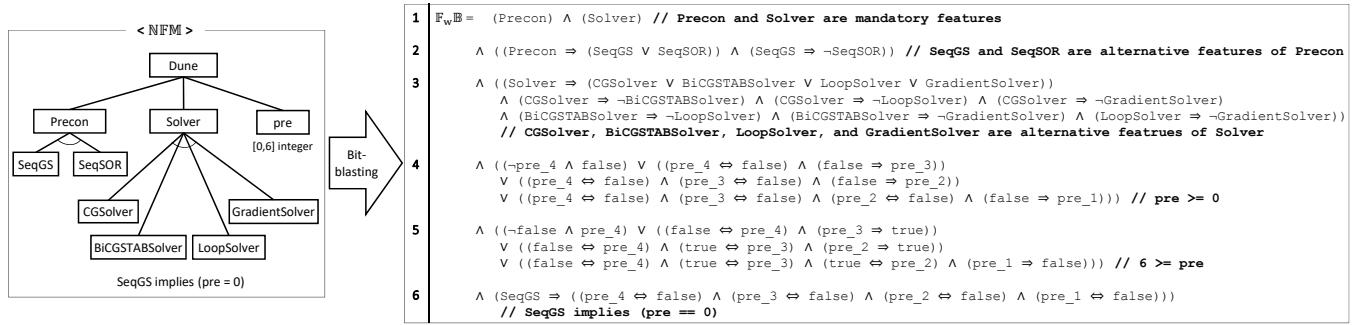


Figure 1: An example of NFM to BBPF Conversion

match the bit-width of the other NF. Extending the bit-width does not change an NF’s possible values due to range constraints.

**Mixed Boolean and NF Constraints.** A numerical constraint can be qualified by Boolean features, such as  $a \Rightarrow (b \neq 0)$ , where  $a$  is a Boolean feature and  $b$  is a NF. In this case, the propositional clauses for NF operations can be generated first (eg., let  $\omega$  be the bit-blasted propositional formula of  $(b \neq 0)$ ), which is then substituted into the original formula to yield the result, namely  $a \Rightarrow \omega$ .

A constraint may inhibit a NF from having any value, meaning that the NF is not used and its value is ignored. In such case, a designated value outside the range of the NF can be used to indicate the NF is ignored, enforced by an equals operation.

**Alternative Features.** For a large set of alternative features, representing them as an NF and keeping a map between its values and alternative features may derive a more compact propositional formula. As an extreme case,  $2^n$  alternative features require  $2^n$  variables, while representing them as a single NF requires only  $n$  bits. Regarding the clauses, alternative features requires  $\binom{2^n}{2} + 1$  CNF clauses,<sup>4</sup> while an NF requires none. A NF that allows multiple discrete values (eg., odd numbers {2, 3, 7, 11, 13...}) instead of values within a range can be encoded in the same manner.

Fig. 1 shows our encoding of an NFM as a BBPF. This NFM was taken from the Dune multi-grid solver [32]. Note that some features and constraints are modified for better illustration.

In Fig. 1, the clauses for Boolean features are represented in lines 1–3, while the clauses for NF is conjoined at lines 4–6. As the NF ‘pre’ has range [0, 6], 4 bits are allocated (including ‘pre\_4’ as its sign bit). Lines 4 and 5 specify the range of the ‘pre’ feature. Line 6 encodes the constraint between a Boolean feature and a NF, where bit-blasting clauses for an equality operation has an implication relationship with a Boolean feature ‘SeqGS’.

## 4 EVALUATION

Our work counts and uniform samples configurations of NFMs. We answer the following research questions to evaluate BBPF:

- RQ1 – How many bits per NF are feasible with bit blasting (BB)?
- RQ2 – Does BBPF allow faster counting?
- RQ3 – Does BBPF allow URS?
- RQ4 – Can existing SAT-analyses of SPLs use BBPF?

<sup>4</sup>  $\binom{2^n}{2}$  clauses are need to ensure only one among the alternative feature is selected, while another clause is to ensure at least one among them is selected.

**RQ1** evaluates a scalability metric of bit blasting, while **RQ2** and **RQ3** evaluate how BBPF perform compared to state-of-art SMT and CP solvers. **RQ4** evaluates whether BBPF can be used with existing SAT-analyses for SPLs.

We used real-world NFMs from [38] and [58] that constrain both Boolean and numerical features. Table 2 lists each NFM with its description, where each system has a different number of NFs and/or difference configuration space size. Henceforth, we use **FSE2015** to denote the feature models from Siegmund *et al.* [58].

FSE2015 NFMs have relatively small configuration spaces, but the equation solving times of all the configurations were benchmarked, so that we can rank them. These NFMs were written for the SPLConqueror tool [58], which we have translated into BBPF. Their smallest NFM had range [1,4]; the largest had [66,4096].

Compared to FSE2015 NFMs, KConfig models have many more features and have huge configuration spaces. The KClause tool [58] derives a propositional formula for each KConfig model. As KClause simplifies NFs to have their default values only, we augmented their formula with bit blasting to allow different NF values.

The KConfig NFMs that we examined had NFs as small as [0,1] and as large as [0,  $2^{32}-1$ ]. When the range of a NF is not defined in a KConfig model, any value within the range of the integer data type is possible, which is [0,  $2^{32}-1$ ]. For NFs that exceed the range [0,  $2^{10}-1$ ], we discretized them to have  $2^{10}$  possible values. We benchmarked the build size of each sampled configuration for the performance analysis of **RQ4**.

To generate a propositional formula for an NF and constraints for BBPF, we used the formula printing functionality of the Z3 solver. To count the number of configurations in BBPF, we used sharpSAT [64], a state-of-art model counter for propositional formulas. To sample configurations of a BBPF, we used Smarch [38], a state-of-art tool for URS propositional formula solutions.

### RQ1: How many bits per NF are feasible with BB?

The most complicated numerical constraint that appeared in the systems we analyzed is  $(A+B>C)$ , where  $A$ ,  $B$ , and  $C$  are NFs of unsigned integers. We consider this constraint as an upper bound on the overhead of numerical constraints.<sup>5</sup> Propositional formulas with three  $b$ -bit NFs related by this constraint were generated and

<sup>5</sup> Actual numerical constraints were simpler, as  $C$  was substituted with a constant. Constants in constraints simplifies the formula by Boolean constraint propagation.

Type	NFM	Description	# Features	# NFs	# Configs	Benchmark	Ref.
FSE2015	Dune	Multi-grid solver	11	3	2,304	Equation solving times	[58]
	HSMGP	Stencil-grid solver	14	3	3,456		
	HiPAcc	Image processing framework	33	2	13,485		
	Trimesh	Triangle mesh library	13	4	239,360		
KConfig	axTLS	Client-server library	94	9	$4.96 \times 10^{38}$	Build sizes	[38]
	Fiasco	Real-time microkernel	234	5	$3.06 \times 10^{12}$		
	uClibc-ng	C library for embedded Linux	269	6	$8.20 \times 10^{45}$		

Table 2: List of Models with Numerical Features and Constraints

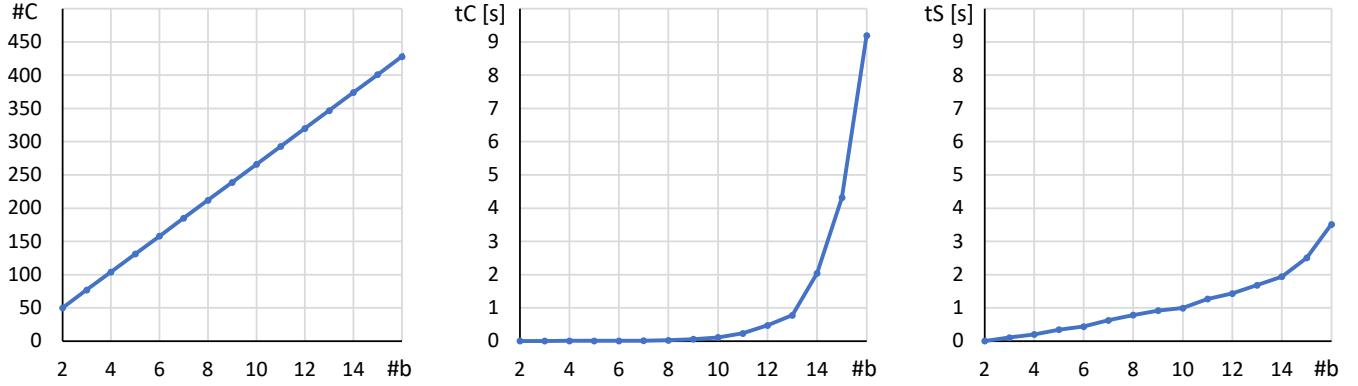


Figure 2: Computation Overhead of the ( $A + B > C$ ) Constraint

benchmarked to determine how many bits are feasible for counting and URS.

Formulas with different bit-widths (#b) from 2 to 32 were generated. For each formula, we measured:

#C – number of CNF clauses in each formula,

$tC$  – time in seconds to count configurations by sharpSAT, and

$tS$  – time in seconds to sample a configuration by Smarch.

To control randomness, we conducted 97 experiments and averaged the results for a confidence level of 95% with a 10% margin of error [61]. If counting or URS took more than 10 seconds, we considered it a *time-out*.

Figure 2 shows our results. The Y-axes show #C,  $tC$ , and  $tS$ ; the X-axes are the number of bits (#b). As  $tC$  timed-out after 16 bits, we show #b up to 16. We observed:

- #C grew linearly with increasing #b,
- $tC$  grew exponentially with increasing #b,  
 $tC$  was below 1 second for  $#b \leq 13$ ,
- $tS$  grew exponentially with increasing #b, and  
 $tS$  was below 1 second for  $#b \leq 10$ .

The linear increase of #C is due to the use of Tseitin's transformation in generating CNF formulas. As the number of NNF variables increases with linearly with #b, Tseitin's transformation guarantees a linear increase  $O(3n+1)$  with the number of variables [65].

$tS$  showed a slower rate of increase compared to  $tC$ , so  $tC > tS$  from  $#b > 14$ . This is due to the formula partitioning of Smarch, which made counting solutions to large formulas faster by counting in a divide-and-conquer manner [38].

These results give a rough idea of the overhead added by NNFs with constraints. The fact that there was a time-out after 16 bits does not mean that NNFs larger than 16 bits cannot be treated by

bit blasting. When a NNF has a value requiring more than 16 bits, we can discretize it to reduce the number of bits to encode it. For example, a 32-bit NNF of range  $[0, 2^{32}-1]$  can be discretized into a 10-bit NNF with the precision of  $2^{22}$ . This makes analyses feasible by reducing the precision of possible values.

**Conclusion: Bit-blasting is feasible up to 16 bits per number (<10 seconds) and has negligible overhead up to 10 bits per number (<1 second).**

## RQ2: Does BBPF allow faster counting?

We compared the time to count solutions using sharpSAT and widely-used SMT and CP solvers. We used Z3<sup>6</sup> as a representative SMT solver and Clafer with the Choco solver<sup>7</sup> as a representative CP solver. Z3 and Clafer use different ways to count the number of configurations than sharpSAT:

- Z3 does not have the functionality to count configurations. A known method involves enumerating the configurations by: 1) deriving a configuration from Z3, 2) making the negation of that solution as a constraint, and 3) repeating 1) and 2) until the constrained model is not satisfiable.<sup>8</sup>
- Clafer has an internal functionality to count configurations, by using the option ‘-n’. Its functionality involves enumerating configurations as well.<sup>9</sup>

<sup>6</sup>Z3py, <https://github.com/Z3Prover/z3>.

<sup>7</sup>Clafer, <https://www.clafer.org/>.

<sup>8</sup>Z3 repository developer response on model counting, <https://github.com/Z3Prover/z3/issues/934>.

<sup>9</sup>Clafer Choco solver source code, <https://github.com/chocoteam/choco-solver/blob/master/src/main/java/org/chocosolver/solver/search/strategy/Search.java>.

Type	Model	Z3	Clafer	BBPF
FSE2015	Dune	26.20s	11s	0.01s
	HSMGP	40.70s	14s	0.01s
	HiPAcc	458s	33s	0.01s
	Trimesh	Time-out	2s	0.01s
KConfig	axTLS	Time-out	Time-out	0.01s
	Fiasco	Time-out	Time-out	0.01s
	uClibc-ng	Time-out	Time-out	0.01s

**Table 3: Average Counting Time Comparison**

We measured the time in seconds to count configurations by each tool. To control randomness, we conducted 97 experiments and averaged the results for a confidence level of 95% with a 10% margin of error [61]. If counting took more than 30 minutes, we considered it a *time-out*. Table 3 shows our results.

We observed with BBPF:

- As expected, counting BBPF by sharpSAT was much faster than Z3 and Clafer, as it does not enumerate solutions.
- KConfig NFM s are too large for Z3 and Clafer to enumerate.

**Conclusion: SharpSAT with BBPF counts configurations considerably faster than Z3 and Clafer. Z3 and Clafer were unusable for KConfig models.**

### RQ3: Does BBPF allow URS?

We now ask if BBPF with Smarch, Z3, and Clafer can URS solutions of a NFM. RQ2 showed Z3 and Clafer can generate samples by enumeration but did not reveal if their samples are uniform. We used techniques in prior work to obtain random samples:

- For Z3, we randomly assigned the value for the parameter ‘random\_seed’, which controls the variable selection heuristic [34].
- For Clafer, we set the ‘–search’ option to ‘random’, which randomizes the order and value of variable assignments.<sup>9</sup>

To check if the samples are uniformly distributed, we rely on a theorem from [38, 51]. Order statistics predict that the average rank of samples from URS are evenly distributed across a configuration space. So if  $n$  samples are taken, the configuration space is partitioned into  $n+1$  equal-length intervals *on average*. The normalized rank (in the unit interval) of the  $k^{th}$ -best performing sample is  $\frac{k}{n+1}$  [51].

With this result, we can check whether samples have evenly distributed ranks using the *Kolmogorov-Smirnov (KS)* test [45]. A KS test checks whether two data sets are sampled from the same distribution. We check if the distribution between sampled ranks and expected ranks are equal with 95% confidence.

First, we used the four NFM s from FSE2015. These NFM s had all of their configurations enumerated and benchmarked, allowing us to know the exact rank of the samples. For each FSE2015 NFM and each tool, we sampled 100, 300, and 500 configurations to evaluate randomness with different sample sizes. For each sample set, we derived the KS test result and the time taken to sample a configuration, averaged from the sample set, in seconds.

**FSE2015 Systems.** Rows 1 through 4 in Table 4 (next page) show the average time taken to sample a configuration for each FSE2015 NFM. Table 5 (next page) shows the result of KS test. We observed:

- Z3 and Clafer had fast average sampling times at .03 and .01 seconds; Smarch took more time at .30 secs,
- Smarch passed KS tests for all NFM s and sample sizes, which says that Smarch performs URS with 95% confidence, and
- Z3 and Clafer failed KS tests for some NFM s and sample sizes. This says that the randomization options for Z3 and Clafer do not always achieve URS.

It is unclear what characteristics of a NFM causes Z3 and Clafer samples to be biased. Prior work on feature models with only Boolean features tried a similar approach to produce random solutions using SAT solvers [18, 35], but they too did not demonstrate URS. In contrast, Smarch delivers URS by construction. That is, it creates a 1-to-1 mapping between a random number and a unique configuration via counting. With Z3 and Clafer, counting is infeasible, as RQ2 showed.

**KConfig Systems.** To demonstrate scalability of sampling, we also present the evaluation with KConfig NFM s. Note that, we could not check the randomness of the samples in the same manner a FSE2015 NFM s, as their configuration space cannot be enumerated to obtain the precise rank of selected configurations.<sup>10</sup> Instead, we utilized the evaluation method in [38], to evaluate whether samples from Z3 and Clafer are uniformly distributed using Smarch.

Smarch achieves URS by using a one-to-one mapping between a number and a configuration. When a random number between 1 to the total number of configurations is given, Smarch outputs a corresponding configuration. Smarch also is capable of the inverse operation, so that it outputs the corresponding number of a given configuration. For the samples taken from Z3 and Clafer, we used Smarch to output the numbers and consider them as the rank of those samples. We then evaluated whether those ranks are uniformly distributed using the KS test.

Rows 5 through 7 in Table 4 and Table 5 are the results. KConfig systems shows a similar trend with the FSE2015 systems. Even for models with larger ranged NFM s and larger configuration spaces, Smarch was able to sample configurations within a reasonable time. Samples from Z3 and Clafer failed the KS test for all systems and sample sizes, indicating that these tools are not capable of URS configuration spaces that are large as KConfig. On the other hand, samples from Smarch passed all KS tests, as it used uniform random numbers to generate the configurations.

**Conclusion: Smarch can perform URS of NFM configurations, which Z3 and Clafer cannot guarantee.**

### RQ4: Can existing SAT-analyses of SPLs use BBPFs?

We explained in Section 2.3 how URS can help finding near-optimal configurations. (Recall taking  $n$  samples, benchmarking each selected configuration, and identifying  $c_{best}$  – the best performing configuration, would be in the top  $\frac{1}{(n+1)}$  percentile of all

<sup>10</sup>We could sort all FSE2015 configurations by performance, so finding the performance rank of a given configuration is easy. In KConfig systems, we do **not** know the performance of all configurations – only those that we sample. Hence we can only estimate the performance rank of a given configuration.

Type	# Samples	NFM			Z3			Clafer			Smarch+BBPF		
		100	300	500	100	300	500	100	300	500	100	300	500
FSE2015	Dune	0.02	0.02	0.02	<0.01	<0.01	<0.01	0.25	0.25	0.28			
	HSMGP	0.02	0.02	0.02	<0.01	<0.01	<0.01	0.30	0.31	0.31			
	HiPAcc	0.05	0.05	0.05	<0.01	<0.01	<0.01	0.54	0.54	0.55			
	Trimesh	0.02	0.02	0.02	<0.01	<0.01	<0.01	0.60	0.61	0.61			
KConfig	axTLS	0.12	0.12	0.12	0.02	0.01	0.01	1.26	1.30	1.27			
	Fiasco	0.25	0.24	0.24	0.03	0.01	0.01	1.50	1.49	1.51			
	uClibc-ng	0.28	0.27	0.27	0.05	0.02	0.02	5.56	5.39	5.62			

Table 4: Average Sampling Times Comparison

	NFM	Z3			Clafer			Smarch+BBPF		
	# Samples	100	300	500	100	300	500	100	300	500
FSE2015	Dune	Fail	Pass	Fail	Pass	Pass	Pass	Pass	Pass	Pass
	HSMGP	Pass	Fail	Pass	Pass	Pass	Fail	Pass	Pass	Pass
	HiPAcc	Fail	Fail	Fail	Pass	Pass	Pass	Pass	Pass	Pass
	Trimesh	Fail	Fail	Fail	Pass	Fail	Fail	Pass	Pass	Pass
KConfig	axTLS	Fail	Fail	Fail	Fail	Fail	Fail	Pass	Pass	Pass
	Fiasco	Fail	Fail	Fail	Fail	Fail	Fail	Pass	Pass	Pass
	uClibc-ng	Fail	Fail	Fail	Fail	Fail	Fail	Pass	Pass	Pass

Table 5: KS Test Result on Uniformity of Samples

configurations *on average*.) Oh *et al.* [51] also proposed a recursive searching algorithm called SRS, which recursively: 1) samples configurations, 2) use samples to reason features that improves performance, and 3) constricts the search space with the found features. They demonstrated that SRS performs better than URS alone.

Their work, however, focused on feature models with Boolean features and constraints, while optimizing feature models with NFs and constraints was left as future work. We wanted to see if their work could be used "as is" with BBPF.

We replicated SRS to find near-optimal configurations from the NFM s of RQ2. For FSE2015 NFM s, we tried to find the configuration with the smallest benchmarked performance, which was the "equation solving time" (see [58] for details). For an experiment, we performed SRS with 20 samples per recursion, which was claimed in [51] as a sufficient sample size to make accurate statistical decisions on the features.

From the FSE2015 NFM s, we gathered following metrics per experiment regarding configuration ranks:

$N$  – total number of samples taken by SRS,

$r_{SRS}$  – normalized rank of  $c_{best}$ , found from SRS,

$r_{URS}$  – expected normalized rank of  $c_{best}$  from  $N$  configurations by URS. It is derived from order statistics, as  $\frac{1}{(N+1)}$ .

In addition, we analyzed the performance value of the found configurations from FSE2015 NFM s. Performance values were normalized by the actual best and worst performance value in the configuration space. We measured:

$p_{SRS}$  – normalized performance of  $c_{best}$  found by SRS, and

$p_{URS}$  – normalized performance of the configuration at rank  $r_{URS}$ .

To control randomness, we conducted 97 experiments and averaged the results for a confidence level of 95% with a 10% margin of error [61]. From the experiments, we derived:

$rTest$  – Mann-Whitney U test results, which evaluates whether  $r_{SRS}$  values are smaller than  $r_{URS}$  values with 95% confidence [44]. "Pass" implies  $r_{SRS}$  is smaller, and "Fail" otherwise.<sup>11</sup>

$pTEST$  – Mann-Whitney U test results from 97 experiments which evaluates whether  $p_{SRS}$  values are smaller than  $p_{URS}$  values with 95% confidence [44]. "Pass" implies  $p_{SRS}$  is smaller, and "Fail" otherwise.

$rBetter$  – SRS success rate is the percentage of experiments that SRS outperforms URS, where  $r_{SRS} < r_{URS}$  is expected.

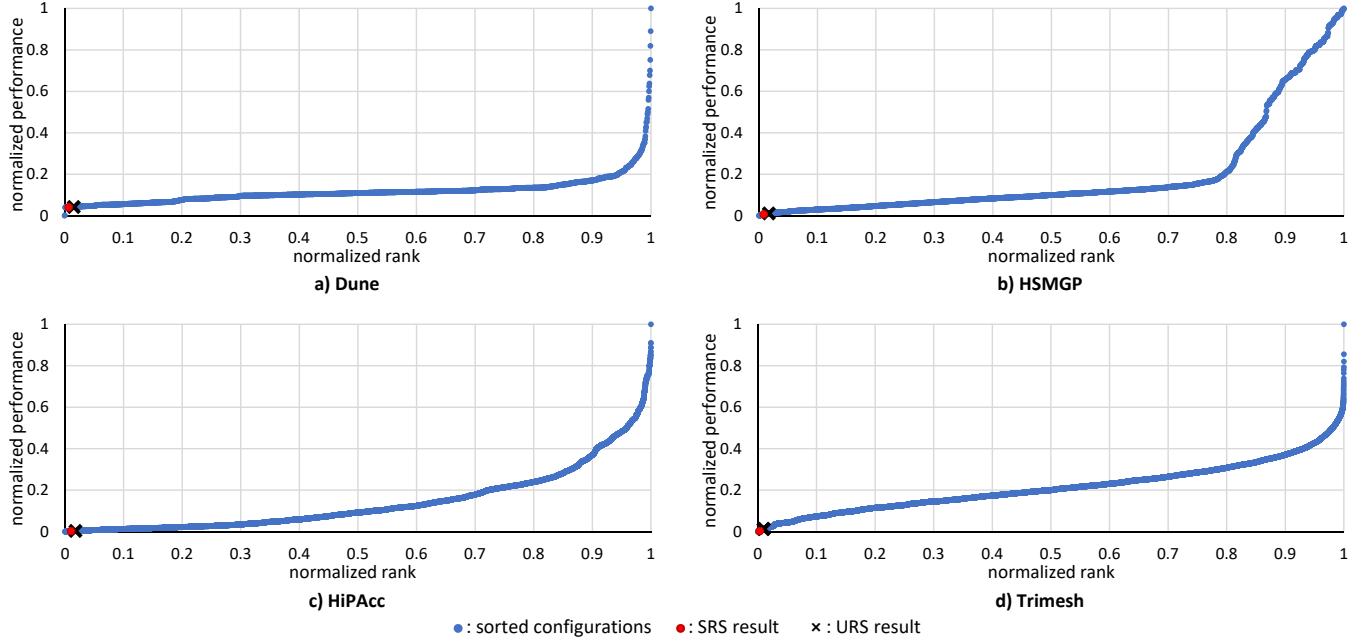
Table 6 shows the rank results for each FSE2015 NFM. We observed:

- The average rank of solutions SRS found were ~.8% away from optimal; the average rank of solutions URS found were ~1.4% away from optimal. Both are good results.
- $N$  was different for all NFM s, as the number of features, constraints, and how a feature affects the objective to optimize are different for each NFM,
- $r_{SRS}$  was lower than  $r_{URS}$  for all NFM s with 95% confidence, which indicates SRS outperforms URS,
- $p_{SRS}$  was lower than  $p_{URS}$  for all NFM s with 95% confidence as well, which also indicates SRS finds better performing configurations than URS, and
- $r_{Better}$  was not 100% in all experiments, meaning that occasionally SRS performs worse than URS. SRS performs better than URS in 89% of all the experiments.

To visualize our results, Figure 3 plots all the configurations of the FSE2015 NFM s, sorted by their performance. The X-axis denotes

<sup>11</sup>We used Mann-Whitney U Test as the distributions of the results are non-parametric.

NFM	<i>N</i>	<i>rSRS</i>	<i>rURS</i>	<i>rTest</i>	<i>pSRS</i>	<i>pURS</i>	<i>pTest</i>	<i>rBetter</i>
<b>Dune</b>	71.32	0.007	0.016	Pass	0.039	0.042	Pass	93%
<b>HSMGP</b>	66.42	0.008	0.017	Pass	0.005	0.011	Pass	91%
<b>HiPAcc</b>	65.82	0.010	0.017	Pass	0.002	0.004	Pass	82%
<b>Trimesh</b>	129.21	0.003	0.009	Pass	0.003	0.013	Pass	91%

**Table 6: Finding Near-Optimal Configurations for FSE2015 Systems****Figure 3: Configuration Space of FSE2015 Systems**

their normalized rank, while the Y-axis denotes their normalized performance. The red dot (•) indicates the configuration found from SRS, while the black × symbol indicates the expected configuration from URS.

Figure 3 shows the configuration found from both SRS and URS are very close to the actual best configuration, regarding both X and Y axis. One exception is Dune, where the best two configurations had much better performance compared to all others. SRS was able to find the two configurations for some experiments, but not always.

We performed similar experiments with KConfig NFM to find configurations with the smallest build size. Since the configuration space cannot be enumerated, we do not know what the best and worst performance values are as well as the rank of the returned configurations. At least, we compared the non-normalized build size of configurations found from SRS to that of URS. To do so, we limited *N* to 200 and derived:

*pSRS* – smallest build size in megabytes, found from SRS with 200 samples, and

*pURS* – smallest build size in megabytes, found from the configurations in 200 URS.

We repeated the experiment 25 times and averaged the result for a confidence level of 95% with 20% margin of error [61]. From these experiments, we derived:

*pTest* — Mann-Whitney U test results which evaluates *pSRS* values are smaller than *pURS* values with 95% confidence [44]. "Pass" implies *pSRS* is smaller, and "Fail" otherwise.

Table 7 shows our results for each NFM.

NFM	<i>pSRS</i>	<i>pURS</i>	<i>pTest</i>
<b>axTLS</b>	0.23	0.24	Pass
<b>Fiasco</b>	25.64	26.74	Pass
<b>uClibc-ng</b>	1.10	1.32	Pass

**Table 7: Finding Near-Optimal Configs for KConfig Systems**

For all systems, we observed that SRS finds configurations with smaller build sizes with 95% confidence. Although the actual rank of configurations sampled is unknown, the results are consistent with FSE2015 NFM as: 1) *rURS* does not depend on the size of the configuration space, but how many samples are collected, and 2) *pSRS* < *pURS*, which corresponds to *rSRS* < *rURS*.

These results show that SRS can perform accurate statistical reasoning over numerical features as well, while also showing that BBPF allows SRS to deal with numerical features without

modifying the algorithm or the solver it uses. However, we believe that SRS can be enhanced to derive more accurate reasoning on numerical features, which may increase the *rPass* value. We leave this as a future work.

**Conclusion:** **BBPF** can be used by existing SAT analysis on SPLs "as is", with the work of [51] as an example.

## Threats to Validity

**Internal validity.** To control randomness, we conducted 97 experiments and averaged the results for a confidence level of 95% with a 10% margin of error [61]. One exception is the result for KConfig NFM in **RQ4**, which repeated the experiments 25 times for 95% confidence and 20% margin of error [61], due to the lengthy time to build sampled configurations.

For **RQ2** and **RQ3**, we utilized the method for counting and URS configurations that are either proposed by the developers of the tool or practiced by prior work in SPL research.

For **RQ4**, we reduced the noise on the performance measurement of the samples as much as possible. FSE2015 NFMs use the performance measurement from [32], which was used in prior works as well. KConfig NFMs measured build sizes, as they are less susceptible to environmental influences.

**External validity.** We used 7 real-world systems with different numbers of features, number of clauses, and domains. Systems had different combinations of constraints with each other, so that we could evaluate our approach with different complexity of NFMs. We are aware that our results may not generalize to all SPLs. At least, our results show identical trends across systems, which provides confidence that our conclusions should hold for many SPLs with comparable size of the configuration space.

We are also aware that Z3 and Clafer may not be representative of all CMT and CP solvers. At least, we used the tools that were widely used in SPL research, which are likely to be used in future SPL research as well.

## 5 RELATED WORK

Adding to Section 2, we discuss other relevant work here.

### 5.1 NFMs in NFMs

Most papers, for various reasons, did not describe how numerical variables were represented as features. Some considered NFMs in the same manner as mandatory Boolean features, so that they had only one value [11, 38]. Some encoded NFMs as alternative features, where each value of a NF was considered a distinct feature [41]. Shi [57] used a single type of feature called 'pseudo-Boolean features'. In his work, Successor (+1) and Predecessor (-1) were introduced as a new type of constraint. As described in Section 3, representing alternative features as a propositional formula has limited scalability as the number of clauses grow rapidly as number of features increases.

Numerical variables and string-attributed feature models have been formalized. *Extended, Advanced or Attributed feature models* appear in the literature as a way to expand classical feature models. Attributed feature models extend Boolean feature models to include additional information about features [8, 10, 54]. In these works,

the authors represent packages of attributes (eg., cost, performance) bound to every Boolean feature in the extended feature model. Those attributes are not NFs [59]. The main differences between attributes and NFs are:

- Currently, there is no consensus on a notation to define attributes. However, most proposals agree that an attribute should consist at least of a name, a domain and a value [8], while a NF consists of a name and a domain [40].
- A NF is a feature, so it can be selected or deselected; it can have a value of zero, or it can have any value, and all these states are different. An attribute, in contrast, cannot be selected/un-selected [8].
- Every Boolean feature in an extended feature model is associated with a set of attributes [40]. A NF in a NFM has a parent, and is affected by cardinality relationships [25].
- A set of attributes can contain several variables. Additionally, those variables can be present in different sets at the same time, as their respective value can be distributed along several sets belonging to different features [8]. Instead, Boolean and NFs are declared just once within the Feature Model [21].
- If we modify the value of just one NF in a configuration, we are producing another configuration in the configuration space. That does not happen with attributes [40].

In any case, constraints are similarly formalized for both NFs and attributes [8].

### 5.2 Automated Reasoning of SPLs

As SPLs have many features and complex constraints, automated solvers were used to solve them as *Constraint Satisfaction Problems (CSP)*. SAT, SMT, CP and *Binary Decision Diagrams (BDDs)* can be considered as different types of CSPs.

For classical feature models, SAT and BDD were the two most utilized automated solvers. For both, a feature model needs to be encoded as a propositional formula. SAT solvers and BDDs were utilized in various analyses, including: checking if a feature model has conflicting constraints or deriving a valid configuration [7, 63], analyzing the structure of the FM [11, 31, 42], counting number of valid configurations [51, 63], and finding inconsistencies between code and feature model [41, 50]. SAT solvers and their variants, such as MaxSAT and #SAT solvers, were used to count the number of configurations and generate samples for testing and finding optimal configurations [18, 35, 38, 58].

For NFMs, SMT and CP solvers were used as they natively support representation and reasoning of NFs and constraints. Encoding feature models for SMT and CP solvers is similar to that of a SAT solver except that they allow numerical variables and operations. Each variable represents a feature and constraints are represented with logical or arithmetic operations. As SMT and CP solvers have similar functionality as SAT solvers, they had similar usage in SPL research: finding conflicts between constraints [9, 21, 46], deriving valid configurations under user-imposed constraints [48], and generating samples for finding optimal configurations [34, 56, 58].

### 5.3 Solvers using Bit-Blasting

Bit-blasting can, computationally speaking, exhaust any solver if the input formula contains numerical values with large bit-width

or complex arithmetic. Then, a pre-processing and simplification of the input formula is essential for reasoning efficiency.

In [20], the authors describe several classes of simplification methods implemented in the solver MathSAT<sub>5</sub>, which are applied with certain heuristics like canonization (eg.,  $X - X = 0$ ), unconstrained propagation, packet splitting [5] and disjunctive partitioning [16] (*i.e.*, the formula is increasingly processed in batches). Approaches like MathSAT<sub>5</sub> are elegant, but are restricted to a subset of bit-vector arithmetic comprising concatenation, extraction, and linear equations over bit-vectors; inequalities are not considered [15]. Although bit-vector theory admits quantifier elimination by considering that a fix-width is the maximum-width among all variables, this is rarely a practical approach. Instead, equisatisfiable formulas are used [39].

Solvers Z3 [22] and Yices [23] apply bit-blasting to every operation besides equality, which is, then, handled by a specialized solver. They also add axioms, dynamically, from array theory. Boolector [14] applies bit-blasting to bit-vector operations and lazily instantiates definitions of array axioms and macros.

A more recent solver is CVC4 [4]. It is a lazy and layered solver, which tries to decide satisfiability using faster, but incomplete, sub-solvers for inequality reasoning. In case of sub-solvers are not enough, theory lemmas and propagated literals are added to the formula, and a lazy CNF-SAT bit-blasting solver is employed. STP [30] performs several array optimizations, as well as arithmetic and Boolean simplifications on the bit-vector formula before bit-blasting to MiniSat [60].

#### 5.4 Uniform Sampling of SPLs

URS is not simple, as merely random selecting features rarely yield valid configurations [43]. Chakraborty et al. [17] proposed Unigen2, a uniform sampling algorithm for propositional formula based on an approximate #SAT solver. Dutra et al. [24] proposed QuickSampler, a sampling algorithm for efficiently generating valid configurations for testing. On these algorithms, Plazar et al. [52] showed that Unigen2 is not scalable for configuration spaces larger than  $10^{10}$ , which is not applicable for our Kconfig NFM, while QuickSampler samples are often not uniformly distributed.

We used Smarch [38], a URS algorithm that can scale up to configuration spaces of size  $10^{249}$ .

#### 5.5 Statistical Analyses of SPLs

Prior work on SPLs performed statistical analyses to reason on colossal ( $\gg 10^{82}$ ) and complex configuration spaces. To estimate the influence of a feature on performance, samples were benchmarked and compared for performance differences [33, 55]. To find optimal configurations, samples were used to search the configurations throughout the space [18, 26, 34, 35, 51, 56]. To evaluate different sampling approaches to locate variability bugs, URS was considered to be the baseline to compare with other approaches [1, 47, 62].

### 6 CONCLUSIONS AND FUTURE WORK

Configuration spaces grow exponentially with increasing number of features, which makes statistical reasoning crucial for understanding them. Compared to classical feature models, NFM have comparatively larger and more complex configuration spaces due to

increased variability and additional types of constraints. This makes statistical reasoning of NFM even more vital. Well-known automated solvers that handle numerical variables, however, were not feasible for counting and URS of configurations for NFM, which are needed for unbiased statistical reasoning of product spaces.

We evaluated bit blasting to encode NFs and their constraints as propositional formulas, to utilize existing SAT-based approaches on counting and URS configurations. With bit blasting, NFM were represented as binary bits while their constraints were represented as propositional clauses.

Our experiments showed bit-blasting:

- can represent NFs and their constraints up to 10 bits of accuracy without overhead,
- can utilize sharpSAT to count the number of configurations, which was much faster and more scalable than current SMT and CP solvers,
- can utilize Smarch [38], an existing tool to URS configurations, while SMT and CP could not guarantee the uniform distribution of their produced samples, and
- was able to use SRS [51], a previously published algorithm, to find near-optimal configurations for classical feature models, to search for near-optimal configurations in NFM as well, and
- the largest KConfig NFM that we examined had a huge configuration space  $10^{45}$  (see Table 2); we believe much larger configuration spaces can be analyzed.

We are confident our work can be utilized by others to analyze different SPLs with NFs. Our research also suggests future explorations:

- expand bit-blasting to handle more arithmetic operations,
- evaluate whether other prior work on analyzing classical feature models with SAT solvers can be extended by bit-blasting.

### ACKNOWLEDGMENTS

Work by Munoz, Pinto and Fuentes is supported by the projects MAGIC P12-TIC1814, TASOVA MCIU-AEI TIN2017-90644-REDT, HADAS TIN2015-64841-R and MEDEA RTI2018-099213-B-I00 (last two co-financed by FEDER funds). Work by Oh and Batory is supported by NSF grants CCF-1212683 and ACI-1550493. This work has been supported by IMFAHE Foundation-Fellowship of 2018

### REFERENCES

- [1] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. InCLing: efficient product-line testing using incremental pairwise sampling. In *ACM SIGPLAN Notices*, Vol. 52. ACM, ACM, ACM, 144–155.
- [2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-oriented software product lines*. Springer, NY, USA.
- [3] Clark Barrett. 2013. Decision Procedures: An Algorithmic Point of View, by Daniel Kroening and Ofer Strichman, Springer-Verlag, 2008. *Journal of Automated Reasoning* 51, 4 (2013), 453–456.
- [4] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification*. Springer, Springer, NY, USA, 171–177.
- [5] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Splitting on demand in SAT modulo theories. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, Springer, NY, USA, 512–526.
- [6] Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer, NY, USA, 305–343.

- [7] Don Batory. 2005. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*. Springer, Springer, NY, USA, 7–20.
- [8] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–636.
- [9] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Using Java CSP solvers in the automated analyses of feature models. In *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, Springer, NY, USA, 399–408.
- [10] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2006. A first step towards a framework for the automated analysis of feature models. *Proc. Managing Variability for Software Product Lines: Working With Variability Mechanisms* 0 (2006), 39–47.
- [11] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1611–1640.
- [12] Armin Biere, Marijn Heule, and Hans van Maaren. 2009. *Handbook of satisfiability*. Vol. 185. IOS press, IEEE.
- [13] Angelo Brillout, Daniel Kroening, and Thomas Wahl. 2009. Mixed abstractions for floating-point arithmetic. In *2009 Formal Methods in Computer-Aided Design*. IEEE, IEEE, Piscataway, NJ, USA, 69–76.
- [14] Robert Brummayer and Armin Biere. 2009. Boolector: An efficient SMT solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Springer, NY, USA, 174–177.
- [15] Randal E Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A Seshia, Ofer Strichman, and Bryan Brady. 2007. Deciding bit-vector arithmetic with abstraction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Springer, NY, USA, 358–372.
- [16] Gianpiero Cabodi, Paolo Camurati, Luciano Lavagno, and Stefano Quer. 1997. Disjunctive partitioning and partial iterative squaring: an effective approach for symbolic traversal of large circuits. In *Proceedings of the 34th annual Design Automation Conference*. ACM, ACM, New York, NY, USA, 728–733.
- [17] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. 2015. On Parallel Scalable Uniform SAT Witness Generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 304–319.
- [18] Jianfeng Chen, Vivek Nair, Rahul Krishna, and Tim Menzies. 2018. "Sampling" as a Baseline Optimizer for Search-based Software Engineering. *IEEE Transactions on Software Engineering* 0 (2018), 12.
- [19] Maciej Ciesielski, Walter Brown, and André Rossi. 2013. Arithmetic bit-level verification using network flow model. In *Haifa Verification Conference*. Springer, Springer, NY, USA, 327–343.
- [20] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The mathsat5 smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Springer, Berlin, Heidelberg, 93–107.
- [21] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2013. Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE, IEEE, 472–481.
- [22] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Springer, NY, USA, 337–340.
- [23] Bruno Dutertre. 2014. Yices 2.2. In *International Conference on Computer Aided Verification*. Springer, Springer, NY, USA, 737–744.
- [24] Rafael Dutra, Kevin Laeuffer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient sampling of SAT solutions for testing. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, ACM, New York, NY, USA, 549–559.
- [25] Holger Eichelberger and Klaus Schmid. 2013. A systematic analysis of textual variability modeling languages. In *Proceedings of the 17th International Software Product Line Conference*. ACM, ACM, New York, NY, USA, 12–21.
- [26] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2017. An empirical study of configuration mismatches in linux. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*. ACM, ACM, New York, NY, USA, 19–28.
- [27] Moshe Emmer, Zurab Khasidashvili, Konstantin Korovin, and Andrei Voronkov. 2010. Encoding industrial hardware verification problems into effectively propositional logic. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc, Springer, NY, USA, 137–144.
- [28] The Linux Foundation. 2018. Kconfig language specification. <https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt>.
- [29] The Linux Foundation. 2018. Kconfig tool specification. <https://www.kernel.org/doc/Documentation/kbuild/kconfig.txt>.
- [30] Vijay Ganesh and David L Dill. 2006. System description of STP.
- [31] Paul Gazzillo. 2017. Kmax: finding all configurations of Kbuild makefiles statically. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, ACM, ACM, 279–290.
- [32] Alexander Grebhahn, Sebastian Kuckuk, Christian Schmitt, Harald Köstler, Norbert Siegmund, Sven Apel, Frank Hannig, and Jürgen Teich. 2014. Experiments on optimizing the performance of stencil codes with spl conqueror. *Parallel Processing Letters* 24, 03 (2014), 1441001.
- [33] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware Performance Prediction: A Statistical Learning Approach. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, Piscataway, NJ, USA, 301–311. <https://doi.org/10.1109/ASE.2013.6693089>
- [34] Jianmei Guo, Jia Hui Liang, Kai Shi, Dingyu Yang, Jingsong Zhang, Krzysztof Czarnecki, Vijay Ganesh, and Huiqun Yu. 2017. SMTIBEA: a hybrid multi-objective optimization algorithm for configuring large constrained software product lines. *Software & Systems Modeling* 0 (2017), 1–20.
- [35] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. 2015. Combining multi-objective search and constraint solving for configuring large software product lines. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. IEEE Press, IEEE/ACM, Piscataway, NJ, USA, 517–528.
- [36] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. 2015. Combining Multi-objective Search and Constraint Solving for Configuring Large Software Product Lines. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 517–528. <http://dl.acm.org/citation.cfm?id=2818754.2818819>
- [37] Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2018. Variability models for generating efficient configurations of functional quality attributes. *Information and Software Technology* 95 (2018), 147–164.
- [38] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. 2019. *Uniform Sampling from Kconfig Feature Models*. Technical Report TR-19-02. University of Texas at Austin, Department of Computer Science.
- [39] Martin Jonas. 2016. *SMT Solving for the Theory of Bit-Vectors*. Ph.D. Dissertation. Masaryk University.
- [40] Ahmet Serkan Karataş, Halit Oğuztün, and Ali Doğru. 2013. From extended feature models to constraint logic programming. *Science of Computer Programming* 78, 12 (2013), 2295–2312.
- [41] Christian Kästner, Paolo G Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *ACM SIGPLAN Notices*, Vol. 46. ACM, IEEE/ACM, Piscataway, NJ, USA, 805–824.
- [42] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. 2015. SAT-based analysis of large real-world feature models is easy. In *Proceedings of the 19th International Conference on Software Product Line*. ACM, IEEE/ACM, Piscataway, NJ, USA, 91–100.
- [43] Jörg Liebig, Alexander Von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, ACM, New York, NY, USA, 81–91.
- [44] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* 0, 0 (1947), 50–60.
- [45] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951), 68–78.
- [46] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2017. Anomaly Detection and Explanation in Context-Aware Software Product Lines. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume B*. ACM, IEEE/ACM, Piscataway, NJ, USA, 18–21.
- [47] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, IEEE/ACM, Piscataway, NJ, USA, 643–654.
- [48] Daniel-Jesús Muñoz, José A. Montenegro, Mónica Pinto, and Lidia Fuentes. 2019. Energy-aware environments for the development of green applications for cyber-physical systems. *Future Generation Computer Systems* 91 (2019), 536 – 554. <https://doi.org/10.1016/j.future.2018.09.006>
- [49] Daniel-Jesús Muñoz, Mónica Pinto, and Lidia Fuentes. 2018. Finding correlations of features affecting energy consumption and performance of web servers using the HADAS eco-assistant. *Computing* 100, 11 (01 Nov 2018), 1155–1173. <https://doi.org/10.1007/s00607-018-0632-7>
- [50] Sarah Nadi and Ric Holt. 2012. Mining Kbuild to detect variability anomalies in Linux. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, IEEE/ACM, Piscataway, NJ, USA, 107–116.
- [51] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, IEEE/ACM, Piscataway, NJ, USA, 61–71.

- [52] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *ICST 2019 - 12th International Conference on Software Testing, Verification, and Validation*. HAL-Inria, Xian, China, 1–12. <https://hal.inria.fr/hal-01991857>
- [53] Francesca Rossi, Peter Van Beek, and Toby Walsh. 2006. *Handbook of constraint programming*. Elsevier, Elsevier.
- [54] Luis Emilio Sánchez, Jorge Andrés Diaz-Pace, and Alejandro Zunino. 2019. A family of heuristic search algorithms for feature model optimization. *Science of Computer Programming* 172 (2019), 264 – 293.
- [55] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-efficient sampling for performance prediction of configurable systems (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, IEEE/ACM, Piscataway, NJ, USA, 342–352.
- [56] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar. 2013. Scalable product line configuration: A straw to break the camel's back. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, ASE, 465–474. <https://doi.org/10.1109/ASE.2013.6693104>
- [57] K. Shi. 2017. Combining Evolutionary Algorithms with Constraint Solving for Configuration Optimization. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE/ACM, Piscataway, NJ, USA, 665–669. <https://doi.org/10.1109/ICSME.2017.32>
- [58] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence Models for Highly Configurable Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 284–294. <https://doi.org/10.1145/2786805.2786845>
- [59] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosemüller, and Gunter Saake. 2012. Predicting Performance via Automated Feature-interaction Detection. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 167–177. <http://dl.acm.org/citation.cfm?id=2337223.2337243>
- [60] Niklas Sorensson and Niklas Een. 2005. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT* 2005, 53 (2005), 1–2.
- [61] Creative Research Systems. 2019. Sample Size Calculator. <https://www.surveysystem.com/sscalc.htm>.
- [62] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2011. Configuration Coverage in the Analysis of Large-scale System Software. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS '11)*. ACM, New York, NY, USA, Article 2, 5 pages. <https://doi.org/10.1145/2039239.2039242>
- [63] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85.
- [64] Marc Thurley. 2006. sharpSAT—counting models with advanced component caching and implicit BCP. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 424–429.
- [65] G. S. Tseitin. 1983. *On the Complexity of Derivation in Propositional Calculus*. Springer Berlin Heidelberg, Berlin, Heidelberg, 466–483. [https://doi.org/10.1007/978-3-642-81955-1\\_28](https://doi.org/10.1007/978-3-642-81955-1_28)
- [66] Christoph M Wintersteiger, Youssef Hamadi, and Leonardo De Moura. 2013. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design* 42, 1 (2013), 3–23.

## A ARTIFACT INFORMATION

The artifact for this paper contains a *Virtual Machine (VM)* with pre-built and configured tools to re-run the evaluation, including: Bit-blasting for NFMFs, model-counting, URS, and SRS for three different types of solvers. A VM is provided due to the amount of knowledge and time necessary to configure and run the third-party and new tools. A Linux operating system is mandatory to run those tools, while a VM can run on almost any operating system and/or hardware. It also includes the tools that natively supports NMFs - Clafer and Z3py. Tested feature models and their intermediate and final results are also included.

### A.1 Access and Content

A VM is pre-configured to re-create the experiments, as well as to reuse for different NFMFs and/or data-sets. The VM and its detailed instructions are available at:

<https://github.com/danieljmg/SPLs-BitBlasting-URS>

The VM make use of the following **third-party** assets:

- Lubuntu 18.04 LTS x86\_64 operating system <sup>12</sup>.
- The Python Interpreter version 3.7 <sup>13</sup>.
- The Oracle's open-source Java Development Kit <sup>14</sup>.
- Clafer Instance Generator 0.45 <sup>15</sup>.
- The Z3 theorem prover SMT solver for Python (Z3py) <sup>16</sup>.
- Model counting SAT solver (SharpSAT) <sup>17</sup>.
- JetBrains 2019.1 Python IDE (PyCharm) <sup>18</sup>.
- The Smarch random sampling tool [38].
- The Kolmogorov-Smirnov Test (KS-t) <sup>19</sup>.
- The Mann-Whitney U Test (MWU-t) <sup>20</sup>.
- The Oracle VirtualBox virtualization software <sup>21</sup>.
- A KConfig measurement VM <sup>22</sup>.

The VM includes the following **new** assets:

- Clafer, Z3py and DIMACS (SharpSAT format) NFMFs of the seven SPLs in Table 2.
- A Python script to transform numerical features modeled in Z3py into Tseitin-CNF DIMACS format using Bit Blasting. It supports composed first order logic and linear arithmetic with integers as in Table 1.
- Scripts to count the number of configurations from Clafer, Z3py and DIMACS models.
- Scripts to random sample configurations from Clafer, Z3py and DIMACS models.
- A Python script to rank sets of random samples to evaluate their uniform distribution. A set of samples is obtained and measured from different reasoners.

<sup>12</sup><https://lubuntu.net/>

<sup>13</sup><https://www.python.org/>

<sup>14</sup><https://openjdk.java.net/>

<sup>15</sup><https://github.com/gsdlab/claferIG>

<sup>16</sup><https://github.com/Z3Prover/z3>

<sup>17</sup><https://github.com/marthurley/sharpSAT>

<sup>18</sup><https://www.jetbrains.com/pycharm>

<sup>19</sup>[https://www.wessa.net/rwasp\\_Rddy-Moores%20K-S%20Test.wasp](https://www.wessa.net/rwasp_Rddy-Moores%20K-S%20Test.wasp)

<sup>20</sup><https://www.sosccsstatistics.com/tests/mannwhitney/default2.aspx>

<sup>21</sup><https://www.virtualbox.org/>

<sup>22</sup>[https://github.com/paulgazz/kconfig\\_case\\_studies](https://github.com/paulgazz/kconfig_case_studies)

<sup>23</sup><https://www.virtualbox.org/wiki/Downloads>

<sup>24</sup><http://www.gnumeric.org/>

- Intermediate files including sets of samples, ranks, and measurements.

### A.2 Installation and Environment Overview

Running the evaluation has following minimum requirements:

- A machine with at least 4GB of memory RAM and 10GB of disk free space, with x86 64-bit operating system and Oracle VirtualBox 6 installed.<sup>23</sup>
- Intel VT-x or AMD-V CPU option activated in the motherboard BIOS settings. However, RQ4 is partially not compatible with Intel VT-x.

To set up the environment, you first need to load the downloaded VM into VirtualBox clicking *File->Import Appliance* and searching for *SPLC19VM.ova*. Lubuntu credentials are:

- **User:** caosd
- **Password and Sudoers password:** splc19

After Lubuntu is ready to use, in its *Desktop* we can find:

- Folder *featuremodels* where all NFMFs with different formats (Clafer, DIMACS and Z3py) are located.
- Folder *samples* where 100, 300 and 500 pre-computed samples for each solver and NFMFs are located.
- Folder *UFscripts* where scripts for model count and sampling with SharpSAT are located.
- Launcher for PyCharm Python IDE.
- Launcher for LXTerminal in order to execute scripts.

### A.3 Usage Summary

Different scripts are provided for each research question (RQ):

- **RQ1:** Open PyCharm and run *Z3toCNF.py* to Bit-blast (*a+b > c*), and finish with *UFscripts/SharpSAT\_ABGTC* scripts.
- **RQ2:** Run the scripts at *UFscripts/SharpSATCounting*, *UFscripts/ClaferCounting* and *PyCharm/Z3*.
- **RQ3:** Run the scripts at *UFscripts/ClaferSampling*, *PyCharm/Z3*, *PyCharm/ranksampling*, *HCS\_Optimizer/randtest.py* and *HCS\_Optimizer/evaluation.py*. Finish with the KS-Test.
- **RQ4:** Adjust and run */search.py* and, for Kconfig models, use the *KConfig measurement* VM ending with the *MWU-test*.

Adjustments required for each RQ is indicated in the code. Data comparisons and graphs can be performed with the included software *Gnumeric*<sup>24</sup>. Detailed steps can be found in the *Github* artifact's page.

# Automated Analysis of Feature Models: Quo Vadis?

José A. Galindo  
Universidad de Sevilla  
Sevilla, Spain  
jagalindo@us.es

David Benavides  
Universidad de Sevilla  
Sevilla, Spain  
benavides@us.es

Pablo Trinidad  
Universidad de Sevilla  
Sevilla, Spain  
ptrinidad@us.es

Antonio-Manuel  
Gutiérrez-Fernández  
ISIS Papyrus  
Brunn am Gebirge, Austria  
antonio.gutierrez@isis-papyrus.com

Antonio Ruiz-Cortés.  
Universidad de Sevilla  
Sevilla, Spain  
aruiz@us.es

## ABSTRACT

Feature models have been used since the 90's to describe software product lines as a way of reusing common parts in a family of software systems. In 2010, a systematic literature review was published summarizing the advances and settling the basis of the area of Automated Analysis of Feature Models (AAFM). From then on, different studies have applied the AAFM in different domains.

AAFM has been applied in different activities along the Software Product Line (SPL) process such as product configuration and derivation, reverse engineering or SPL testing. As the field evolves, there is a need to evaluate the trends in the area and discover where the AAFM is being applied. Systematic Literature Reviews (SLRs) and Systematic Mapping Study (SMS) are the main techniques used to crawl the knowledge in a scientific area and candidates to discover the aforementioned tendencies. While SLRs are suitable to summarize the state of a research area by providing mostly qualitative information, SMSs focus on providing quantitative information and a categorization of the corpus that enables the identification of trends and research opportunities.

We present a SMS to identify the evolution and trends in the application of the AAFM since 2010. Concretely, we have performed a search on different databases of AAFM-related papers. We selected 423 primary sources (papers) that followed the defined inclusion and exclusion criteria. The primary sources were classified according to different variability facets that were found during the reading and key-wording phase. It is important to remark that before 2010, AAFM was not well defined and it was referenced using an amalgam of names and concepts. Therefore, we consider that in 2010 the concept of AAFM was coined and then used in different domains and scenarios. This paper studies how AAFM has been used since its definition.

First, we found six different variability facets where the AAFM is being applied that define the tendencies: product configuration and

derivation; testing and evolution; reverse engineering; multi-model variability-analysis; variability modelling and variability-intensive systems. We also confirmed that there is a lack of industrial evidence in most of the cases. Finally, we present where and when the papers have been published and who are the authors and institutions that are contributing to the field. We observed that the maturity is proven by the increment in the number of journals published along the years as well as the diversity of conferences and workshops where papers are published.

We also observed that there are only a few industrial and real evidences of the application of AAFM techniques in most of the cases. We detect in detail where and when the papers have been published and who are the authors and institutions that are contributing to the field. We saw that the maturity is proven by the increment in the number of journals published along the years as well as the diversity of conferences and workshops where papers are presented. Finally, we devise some research opportunities and applications in the future as well as synergies with other research areas. We also suggest some synergies with other areas such as cloud or mobile computing among others that can motivate further research in the future.

The reader can find the full text of this paper at <https://doi.org/10.1007/s00607-018-0646-1>

## CCS CONCEPTS

• Software and its engineering → Software product lines;

ACM Reference Format:

José A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés.. 2019. Automated Analysis of Feature Models: Quo Vadis?. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342373>

## ACKNOWLEDGEMENTS

This work has been partially funded by the EU FEDER program, the MINECO project OPHELIA (RTI2018-101204-B-C22); the Juan de la Cierva postdoctoral program; the TASOVA network (MCIU-AEI TIN2017-90644-REDT); and the Junta de Andalucía METAMORFOSIS project.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*SPLC '19, September 9–13, 2019, Paris, France*  
© 2019 Association for Computing Machinery.  
ACM ISBN 978-1-4503-7138-4/19/09...\$15.00  
<https://doi.org/10.1145/3336294.3342373>

# A Kconfig Translation to Logic with One-Way Validation System

David Fernandez-Amoros

UNED

Madrid, Spain

david@issi.uned.es

Ruben Heradio

UNED

Madrid, Spain

rheradio@issi.uned.es

Christoph Mayr-Dorn

JKU Institute for Software Systems Engineering, Johannes Kepler University Linz, Austria  
christoph.mayr-dorn@jku.at

Alexander Egyed

JKU Institute for Software Systems Engineering, Johannes Kepler University Linz, Austria  
alexander.egyed@jku.at

## ABSTRACT

Automated analysis of variability models is crucial for managing software system variants, customized for different market segments or contexts of use. As most approaches for automated analysis are built upon logic engines, they require having a Boolean logic translation of the variability models. However, the translation of some significant languages to Boolean logic is remarkably non-trivial. The contribution of this paper is twofold: first, a translation of the Kconfig language is presented; second, an approach to test the translation for any given model is provided. The proposed translation has been empirically tested with the introduced validation procedure on five open-source projects.

### ACM Reference Format:

David Fernandez-Amoros, Ruben Heradio, Christoph Mayr-Dorn, and Alexander Egyed. 2019. A Kconfig Translation to Logic with One-Way Validation System. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3336294.3336313>

## 1 INTRODUCTION

The creation of system variants is essential in software engineering paradigms: software product lines, software ecosystems, context-aware software, etc. In such paradigms, *Variability Models* (VMs) are frequently used to account for the configurable features of the variants.

Automated analysis of VMs has a long history[4, 18, 19]. For instance, to detect whether a model instance (also known as *product* or *configuration*) is *valid* [1–3, 10, 16, 20, 23, 24, 30, 36, 38, 40] (i.e. it does not violate any inter-feature constraints); to provide explanations about the causes that make an instance invalid in

order to guide the user to solve the problem [1, 3, 8, 12, 13, 20, 30, 33, 34, 38, 42]; to detect *dead features* in the model (i.e. features that cannot be part of any valid instance) [8, 12, 13, 20, 25, 26, 33, 34, 39, 41, 42, 44, 45], etc.

A common approach for VM analysis is translating the model into a Boolean formula, which is then processed with a logic engine. For some VM languages, such as feature models, the translation to logic is straightforward [3]. However, the translation of the Kconfig language has not been adequately described. Several attempts have been made to translate Kconfig code to Boolean logic[5, 6, 14, 21, 27–29, 37, 43].

A Kconfig translation is challenging for two major reasons: first, a formal specification of the KConfig language does not exist, only the various versions of the `kconfig-language.txt`<sup>1</sup> file. Second, the translation requires a paradigm shift from imperative to logic: Using the `conf` application, an application engineer can configure the software project. The feature values are variables that can change from one value to another for as long as the configuration process lasts (e.g., a feature with a false value may change to true later when a select clause for another feature is evaluated). This is in contrast to propositional variables which can only have one value in a particular configuration. This paper provides the following two contributions to research on VM analysis:

- (1) A translation from Kconfig to logic which is complete for Boolean configs. The translation supports language features that have been omitted in some of the related work, such as: *chaining*, *visibility conditions*, *user prompts*, *default values*, etc. Also, no artificial variables are added in the translation.
- (2) Empirical validation of the translation on five real-world projects.

## 2 INTRODUCING KCONFIG

This section provides a walkthrough of the main features of the Kconfig language with an emphasis on their role in the translation process to Boolean logic.

**Configs and symbols.** Figure 1 provides an example of the configuration file for the embedded OS of a hypothetical multimedia device. Kconfig deals with *configs*. A config is a declaration of a *symbol* (e.g., `HAVE_WIFI` on line 1) with a *type* (e.g., `bool` on line 2) and other attributes. The type restricts the possible values available for a symbol. A symbol can be declared multiple times (usually with different attributes) in different configs, as long as the type is the

This work has been supported by (i) the Spanish Ministry of Education and Vocational Training under the projects with reference DPI2016-77677-P and CAS17/00022, (ii) the Austrian Science Fund (FWF): P29415-NBL funded by the Government of Upper Austria; and (iii) Pro2Future: FFG, Contract No. 854184.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336313>

<sup>1</sup><https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt>

same. In Figure 1, for example, USE\_CUDA is declared three times (lines 60, 66 and 71).

```

1 config HAVE_WIFI
2   bool
3   default n
4
5 config STREAMING
6   bool "Streaming ?"
7   select WIFI
8
9 config USB
10  bool "Use USB ?"
11
12 config WIFI
13  bool "Use Wifi ?"
14  if HAVE_WIFI
15  default n
16
17 config WPA
18  bool "Use WPA ?"
19  depends on WIFI
20  default y
21
22 config WEP
23  bool
24  default n if WPA2
25  default y
26
27 menu "Compiler"
28   visible if EXPERT
29   depends on DEVEL
30   config GCC
31   bool "Use GCC ?"
32
33   config CLANG
34   bool
35   default !GCC
36 endmenu
37
38
39 config PROFILE
40
41
42 config TWEAK
43   bool "Tweak ?"
44   default y
45   select PROFILE
46     if KEYBOARD
47
48 choice
49   depends on PROFILE
50   prompt "USER "
51   config STANDARD
52     bool "Standard"
53   config ADVANCED
54     bool "Advanced"
55   config EXPERT
56     bool "Expert"
57 endchoice
58
59 if EXPERT
60   config USE_CUDA
61     bool "USE CUDA ?"
62 end-if
63
64 if ADVANCED &&
65   HAVE_CUDA
66   config USE_CUDA
67     bool "USE CUDA ?"
68     default n
69 end-if
70
71 config USE_CUDA
72   bool
73   default n
74
75 if HAVE_WIFI
76   source "Config.in"
77 end-if

```

**Figure 1: Example Kconfig file**

**Symbol types.** There are four types of configs in the projects evaluated: *bool*, *string*, *int*, and *hex*. Bool configs may have  $n$  and  $y$  values to represent logic *false* and *true*. A common use of string configs is to hold the string value of a set of related Boolean configs. The types *int* and *hex* are essentially equal to *string*. For space limitations, this paper will consider only *bool* configs. In the projects evaluated, no string-like types were used inside the *bool* configs, which means that the translation is complete for these configs.

**User input.** Configs may specify a *prompt* to obtain a value from the user. On line 6, the text *Streaming?* is displayed and a value for STREAMING is requested from the user. The prompt may be guarded by a Boolean expression (e.g. lines 13-14). When the expression evaluates to false, the text is not shown and the user input is not requested. If HAVE\_WIFI is false or unassigned, the default value of “n” will be assigned to WIFI.

**Default values** may be specified for a config. In combination with a prompt, the default value is shown as a suggestion to the user. Without a usable prompt (e.g., lines 22-25) the config takes the *default value* if one exists. If there is no default value, the symbol remains unassigned. A later redeclaration of the same symbol may provide a value for it, in the meantime, logical expressions containing an unassigned value evaluate to false. A default clause consists of a value and an optional Boolean expression (e.g., line 24). If there is more than one default clause, they are evaluated in order. Symbol WEP in line 25 has no prompt, so default clauses are employed. If WPA2 is false, the WEP is assigned true. Otherwise, it is assigned true.

**Direct dependencies, menus, and *if* blocks.** A config declaration may have *direct dependencies*; a dependency is a Boolean expression which must be satisfied in order for the config to be evaluated. If the dependency of a config does not hold, the rest of

the declaration is ignored. For example, the config WPA is only evaluated when WIFI is true. A *menu* is a mechanism for grouping related configs (lines 27-36). It allows imposing dependencies to a group of configs at once. A config inside a menu *inherits* the dependencies of the menu. Menus may be nested and thus inherit the dependencies from their respective parents.

**Visibility conditions** further constraint menu evaluation. The visibility condition acts as menu-wide guard to prompts; instead of adding the guard to each individual prompt, it is specified at the menu level. In the “Compiler” menu, both GCC and CLANG depend on DEVEL. Also, the prompt for GCC is not shown unless EXPERT is true. Another way to add dependencies to a group of configs is through use of an *if-block*. It takes a condition and adds it as an additional dependency to the declarations (configs, menus, choices and other if-blocks) inside the block. In line 59, USE\_CUDA is inside an if-block with EXPERT as condition, which is equivalent to having a “depends on EXPERT” clause.

**Reverse dependencies.** A Boolean config may declare *reverse dependencies* through the use of the *select* construction (e.g. line 7). If the config dependencies are met, the symbol (the selector, here STREAMING) logically implies another one (the selected, here WIFI, so that, if the selector is true, the selected has to be true as well. When the select clause is guarded by an expression (e.g. lines 45-46), the selector and the condition imply the selected. In our example, when TWEAK and KEYBOARD are true, PROFILE is assigned “y”, regardless of the previous value.

**Choices.** This construct enables grouping a set of dependencies and config declarations out of which only exactly one config may become true while the rest become false (lines 48-57). If the dependencies do not hold (line 49), then all the choice members are set to false. The choice construction contains config declarations and may also include if-blocks. In the example, if PROFILE is true, the user is asked to choose a value among STANDARD, ADVANCED or EXPERT.

**Importing Kconfig files.** Finally, Kconfig enables including another Kconfig file via the *source* construction (e.g. line 76). If it is inside a menu or if-block, the source will inherit the corresponding dependencies. A source command with unmet dependencies is ignored during the configuration process, i.e. the source file is not included.

### 3 CLARIFYING KCONFIG SEMANTICS

The Kconfig interpreter requires configuration files with correct syntax, although developers often produce incorrect Kconfig code because of false assumptions [17].

Kconfig semantics, however, is primarily defined by what the interpreter accepts as correct. The `kconfig-language.txt` file is insufficient for translating Kconfig to Boolean logic. In the following subsection we describe aspects that have been incorrectly or incompletely described before, that have convoluted semantics, or that tend to lead to incorrect configurations.

### 3.1 Config chaining

Kconfig files typically declare some symbol several times. A new declaration of the same symbol (e.g., lines 59-73 in Fig. 1) does not *override* the previous one. The interpreter evaluates configurations

of the same symbol in order. Every symbol starts out unassigned. Configs of unassigned symbols are evaluated. A config will provide no value for the symbol if either a) its dependencies are not met or b) there is no productive prompt and no productive default (i.e. there are none or they are guarded by conditions that evaluate to false). Instead, the interpreter checks successive declarations of the same symbol as long as it has no value assigned. Once a symbol obtains an assignment (i.e. true or false) all remaining redeclarations are ignored. We coined the term *chaining* to account for this very usual behavior.

### 3.2 Decision overriding with Select

The dependencies of a selected symbol are not evaluated. It is thus possible to set to true a symbol whose dependencies do not hold. The use of the *select* construction is discouraged because it has the potential to produce wrong configurations, bypassing the dependency mechanism. This usage is referred to as *heavy-handed select*. Lines 39-46 in Figure 1 provide an example.

```

1 config H
2   bool "Prompt"
3   select A if !C
4 config J
5   bool
6   default y
7   select D if X
8 config J
9   bool "Prompt"
10  depends on X
11
12  select H
13  config ZA
14  bool "ZA?"
15  select ZB
16  config ZB
17  bool "ZB?"
18  depends on X
19  select ZC
20  config ZC
21  bool

```

Figure 2: Convolved select clauses

### 3.3 Select in chained configs

The interpreter evaluates select statements even in configurations that are not evaluated. The configuration of symbol J in lines 4 to 7 in Figure 2 leads to an assignment of true. While the config of J in lines 8-11 is not evaluated because J is already assigned, the interpreter will nevertheless select H (line 1) if X is true (line 7). The reason is that the interpreter stores select clauses as reverse dependencies of the selected symbols.

### 3.4 Select transitiveness

The documentation is unclear w.r.t. transitivity of select clauses: if ZA selects symbol ZB and ZB selects symbol ZC in Figure 2, does this imply that if ZA is true, then ZC is also true? The answer depends on whether ZB's dependencies are met. The interpreter stores the selector, the dependencies of the config and the optional guard as attributes of the selected. This way, a select clause is only triggered if the dependencies of the selector are true.

## 4 RELATED WORK

One of the first studies of variability models extracted from the Linux kernel was carried out by She et al. [27–29]. The formalism employed uses denotational semantics. There is no translation to logic and no validation attempt is made. She et al. describe the formalism as incomplete since some “corner cases” are not modeled. The only hint as to what are considered corner cases is heavy-handed select. Config declarations are not checked for redeclarations, so chaining of configs is not even considered. This formalism was apparently the source of the translation used by Berger et al. in [5, 6] where a catalog of software system projects was assembled as defined in Kconfig and eCos files, dubbed “Variability Models in

the Wild”. Again no empirical validation of the translation seems to have been performed. There are no details of the translation, although configs are treated separately, so chaining does not seem to be implemented. The models make heavy use of artificial variables, such as those used in the Tseitin construction [35] to transform a formula to CNF.

Tartler et al. [31, 32] adapt the Linux Variability Analysis Tool (LVAT)<sup>2</sup> from She to extract variability models from both Kconfig and code artifacts with the purpose of detecting and repairing inconsistencies between them.

Zengler and Küchlin [43] go a step further and present an encoding into Boolean logic that they illustrate only for the Linux kernel. Their model is updated in [37]. The translation is incomplete; prompts, visibility conditions, and default values are not considered. No experimental validation of the translation is presented. The translation is used to find dead and core features; however chaining is not considered.

In [14], another translation is presented. It is only for Boolean configs and lacks both chaining and validation. The translation is used to build BDDs representing the model.

A thorough experimental revision of the semantics of the Kconfig language is presented in [11]. Nowhere is the issue of chaining mentioned. Neither is the need to add the dependencies of a selector to translate a select clause.

KconfigReader is a tool that translates Kconfig files to Boolean logic. To date, it is the translation covering the most of the Kconfig language. In [21], the validation approach of KconfigReader is described. The translation itself is not explained. The tools can also perform brute-force validation of very small snippets of code using the official conf interpreter capabilities. It provides a useful repository of test cases, some randomly generated and some taken from previous research. Two limitations are mentioned, though: Limitations regarding reverse dependencies and the inability to decide the validity of configurations using string values not mentioned in the kconfig code. The command-line interpreter conf allows checking configurations through command-line options. The process can be summarised as follows: If present, a file containing symbol values from a previous configuration is read, then the configuration process begins. User input is never requested, the values read from the file are used instead. These values may still change (e.g., due to select clauses). When the configuration is finished, the symbol values are written back to the file. Kästner uses this mechanism as an oracle: Every possible combination for the values is generated and written to a file. The interpreter is then invoked on the file. At the end, the file is checked for changes. If the file has not changed, the values are correct, otherwise the values are not valid. KconfigReader includes a repository of test cases, unfortunately, chaining is not even tested. The tool is complete and correct according to our evaluation, although it introduces artificial variables that obscure the meaning.

## 5 TRANSLATION TO BOOLEAN LOGIC

We extended the original parser to extract a set of additional attributes for each declaration: To propagate dependencies and visibility conditions, the parser keeps a stack of open commands (if block,

<sup>2</sup><https://github.com/matachi/linux-variability-analysis-tools.exconfig>

menu, choice) so when a config is declared, the top of the stack is checked and the corresponding attributes inherited. The translation presented here relies heavily on the “if-then-else” construction. Of course, “if A then B else C” can still be considered as a shorthand for the more Boolean logic-looking  $(\neg A \vee B) \wedge (A \vee C)$ .

### 5.1 Config translation

For the translation of bool configs, one Boolean variable with the same name is used. To distinguish different configs of the same symbol a subindex may be used. During the parsing of the Kconfig files, a series of attributes for each declaration are compiled. For a config  $\text{FOO}_i$  a series of syntactic expressions from the parsing phase is derived:  $\text{FOO}_i.\text{dep}$ ,  $\text{FOO}_i.\text{vis}$ ,  $X_i.\text{prompt}$ ,  $\text{FOO}_i.\text{promptGuard}$ , standing for the dependencies of config  $\text{FOO}_i$ , the visibility condition of  $\text{FOO}_i$ , a Boolean value indicating if there is a prompt in the declaration or not, and the expression guarding the prompt (true if there is none), respectively. The system also keeps track of which config selects which symbol for later use. The translation is composed of the following steps: The first piece to put together for each config is a logic formula for the default value. Consider this config:

```
config FOOi
default Vi1 if Ci1
default Vi2 if Ci2
...
default Vini if Cini
```

and let us define:

```
FOOi.default ≡
if Ci1 then Vi1 else
if Ci2 then Vi2 else
...
if Cini then Vini
```

The second aspect to consider is whether the config is selected by another config or not. As mentioned in the previous section, a select clause requires that the selector dependencies are met to activate the selected symbol. So first, a list of selectors is needed, together with their respective dependencies and optional guarding expressions. For  $k \in \{1..m_{\text{FOO}}\}$ , let that be:

```
config SELECTORk
bool ...
select FOO if GUARDk
```

Let us define a formula for later use:

```
FOO.selectCondition ≡
 $\bigwedge_{k=1}^{m_{\text{FOO}}} \text{SELECTOR}_k.\text{dep} \wedge \text{SELECTOR}_k \wedge \text{GUARD}_k$ 
```

Let us now consider the user input. If it is going to be requested, any value is possible for the symbol and nothing else needs to be done. In contrast, if user input is not requested, then each config  $\text{FOO}_i$  is checked for usable default values until a value is computed for the symbol or there are no more configs. For a config  $\text{FOO}_i$  to request user input, the dependencies must hold, and there has

to be a visible prompt meeting its guard. If just one declaration of FOO meets this criterion, user input is asked. So, let us define:

$\text{FOO}.\text{promptCondition} \equiv$

$\bigvee_{i=1}^n (\text{FOO}_i.\text{dep} \wedge \text{FOO}_i.\text{prompt} \wedge \text{FOO}_i.\text{promptGuard})$

To put it all together, the translation for symbol FOO is:

$\text{Translation}(\text{FOO}) \equiv \text{if } \text{FOO}.\text{selectCondition} \text{ then } \text{FOO}$

$\text{else if } \text{FOO}.\text{promptCondition} \text{ then true}$

$\text{else if } \text{FOO}_1.\text{dep} \wedge \bigvee_{i=1}^{n_1} C_{1i} \text{ then } \text{FOO}_1.\text{default}$

$\text{else if } \text{FOO}_2.\text{dep} \wedge \bigvee_{i=1}^{n_2} C_{2i} \text{ then } \text{FOO}_2.\text{default}$

...

$\text{else if } \text{FOO}_r.\text{dep} \wedge \bigvee_{i=1}^{n_r} C_{ri} \text{ then } \text{FOO}_r.\text{default}$

$\text{else } \neg \text{FOO}$

### 5.2 Choice translation

Let us consider a choice involving  $n$  symbols,  $X_1, X_2, \dots, X_n$ . The symbols inside the choice block, also called choice members, are assumed not to be declared anywhere else<sup>3</sup>. The semantics of the Boolean choice is fairly straightforward: if the dependencies hold and the choice is visible, exactly one of the choice members is true. Otherwise, all the choice members must be false. A member of a choice must have a usable prompt to be considered. A choice with no member translates to true. Default values for the choice and selects to choice members are ignored. For each member,  $X_i$ , let us define the auxiliary formula  $B_i \equiv X_i \wedge X_i.\text{prompt} \wedge X_i.\text{promptGuard}$ . The translation for a choice C, would then be:

$\text{translation}(C) \equiv \text{if } C.\text{dep} \wedge C.\text{vis} \wedge C.\text{prompt} \wedge C.\text{promptGuard}$   
 $\wedge \bigvee_{i=1}^n (X_i.\text{prompt} \wedge X_i.\text{promptGuard}) \text{ then}$   
 $\bigvee_{i=1}^n B_i \wedge \bigwedge_{j=1}^n \neg(B_i \wedge B_j) \text{ else } \bigwedge_{i=1}^n \neg B_i$ . The result of the trans-

lation process is a set of formulas whose conjunction provides a logical model for the corresponding Kconfig files.

## 6 ONE-WAY VALIDATION OF THE PROPOSED TRANSLATION

Figure 3 summarizes the one-way validation approach. For a Kconfig translation to Boolean logic, S, a random sample of valid products  $p_1, p_2, \dots, p_n$  is generated using the *conf* program. Then,  $n$  BDDs are built, one for each conjunction  $S \wedge p_1, S \wedge p_2, \dots, S \wedge p_n$ . Each BDD is then checked to see if there is only one solution, namely that of the corresponding product. Otherwise, the logic translation is invalid. BDDs are described in detail in [9, 22]. There are two

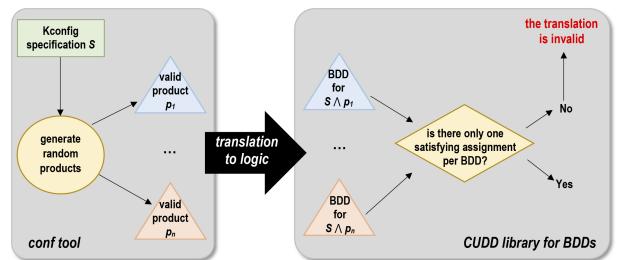


Figure 3: Schema of the one-way validation approach

good reasons to use BDD technology for our validation approach:

<sup>3</sup>The interpreter would otherwise issue a warning.

**1) BDDs do not require CNF-formulas.** A BDD can be built by feeding it formulas which do not need to be in *Conjunctive Normal Form (CNF)*. This is unlike SAT-solvers [7], which traditionally require them. Translation to CNF can produce a formula exponentially longer than the original one (e.g. the translation for freetz in CNF would require billions of clauses). Alternative approaches to get a CNF formula, such as the Tseitin’s construction of an equisatisfiable formula [35], require the introduction of a great deal of artificial variables which impede efficiency and obscure the meaning.

**2) BDD-based validation approach helps debugging the translation.** Getting the translation of a complex VM language to logic right usually requires several iterations. When a generated product does not validate, there are two possibilities: a) There are zero solutions. b) There is more than one solution. If there are no solutions for the BDD, there is a contradiction. In that case, the building process can be retraced to the point in which the BDD became *false*, because the number of nonterminal nodes in the BDD drops to zero. The offending formula points to the symbol(s) or choice that is causing the problem. If there is more than one satisfying assignment, BDDs are more useful than SAT-solvers: In a valid product, there is only one correct value for each variable. For that reason, there is also only one node per variable in the BDD and for every node, one of the children has to point to false. There also no empty levels. It suffices to traverse the BDD, find out which nodes violate that property, and associate them with the variables. This way, a set of symbols whose translation is wrong is obtained, which can be used to troubleshoot the translation.

The biggest disadvantage of BDDs (the potential for memory exhaustion) is mitigated because instead of facing the problem head-on and making a BDD of the full logical model, an alternative route is taken to always build the BDD corresponding to the model and a product to be tested. Such a BDD is known to have only one node per variable, assuming the model is correct.

To validate the translation described in Section 5, the open-source projects in Table 1 have been used as test-bed. These projects are heterogeneous in the number of configs and concerning how they use the Kconfig language, thus being a representative sample to account for the diversity of the population of Kconfig projects. The table also shows that the vast majority of configs were of type bool. A thousand products for each project were generated and all of them validated for the translation presented in the paper (i.e. the conjunction of the model translation plus each individual product has exactly one satisfying assignment). The debugging process allowed us to discover the chaining mechanism and also the need to include a selector’s dependencies in the translation. The code and instructions to replicate the results have been made available in a repository<sup>4</sup>. We also evaluated KconfigReader [21] under the same conditions (i.e. only bool configs). KconfigReader translates all the symbol types as opposed to only bool, but failed to build models for busybox and uClibc due to syntax errors. For the other projects the translation process completed successfully and the models validated without any problems, although the translation involved adding additional variables not corresponding to config definitions, which goes in the way of later product line analysis. Interestingly, the short snippet validation approach in [21] carried over to the big

Name	#Feat.	% bool	Kconfig2Logic**		KconfigReader	
			Time*	Valid	Time*	Valid
toybox	12	83.33	0.04	100%	0.03	100%
axtls	64	67.36	0.05	100%	0.05	100%
uClibc	303	87.82	1.26	100%	error	error
busybox	604	94.52	2.40	100%	error	error
freetz	6492	98.43	69.3	100%	323.8	100%

\*Avg time taken to generate and test one configuration (seconds).

\*\*Our approach

**Table 1: Project validation for 1000 sample products**

projects. The running times comparison shows that our translation scales better than KconfigReader. The reason seems to be that KconfigReader produces a very long translation, while our approach only produces at most one constraint per config. For instance, for the freetz project, KconfigReader produced 81780 constraints vs. 5373 for our approach.

To build the BDDs, the CUDD package by Fabio Somenzi was used<sup>5</sup>. The experiments were performed on an Intel Xeon@2Ghz running Linux. The memory requirements are very small, less than 250MB per run with the biggest model.

## 7 THREATS TO VALIDITY AND LIMITATIONS

The primary threat to validity of our translation is the modification of the *conf* program to produce random valid products. This modification improves the reliability of our validation procedure: while the original *conf* program has an option to create random configurations, it uses different code for generating configurations than for configuring them. This could give rise to incompatible results. To mitigate this problem, *conf* was modified with care not to alter its intended behavior. The interpreter is designed to show the user a list of acceptable values for boolean and tristate types. We reprogrammed the function waiting for user input to return a randomly chosen value among the alternatives to ensure that generating and configuring work the same way. A limitation of the validation is that it goes is one-way only. We make no assurances that the translation is correct the other way around.

## 8 CONCLUSIONS

The translation of a VM to logic is a necessity for most automated analysis approaches because they are built upon logic engines. This paper has described how to translate Kconfig to logic for bool configs, including some of its most obscure constructions, such as config chaining and convoluted selects. We have also checked if valid products according to a Kconfig specification remain valid according to its logical translation for five different projects. Both our translation and KconfigReader passed the tests. KconfigReader is able to translate string types but our approach is more compact, more scalable, more compatible with Kconfig varieties, and overall clearer, since there is a one-to-one correspondence between configs and logical variables. Future work will explain the translation of string and tristate types.

<sup>4</sup><https://figshare.com/s/df2b0e4bc889a701f3f3>

<sup>5</sup><http://vlsi.colorado.edu/~fabio/>

## REFERENCES

- [1] Hai H. Wang A, Yuan Fang Li B, Jing Sun C, Hongyu Zhang D, and Jeff Pan E. 2007. Verifying feature models using OWL. *Journal of Web Semantics* 5 (2007), 117–129.
- [2] Randall C. Bachmeyer and Harry S. Delugach. 2007. A Conceptual Graph Approach to Feature Modeling. In *Conceptual Structures: Knowledge Architectures for Smart Applications, 15th International Conference on Conceptual Structures, (ICCS)*. Springer, 179–191. [https://doi.org/10.1007/978-3-540-73681-3\\_14](https://doi.org/10.1007/978-3-540-73681-3_14)
- [3] Don Batory. 2005. Feature models, grammars, and propositional formulas. In *9th international conference on Software Product Lines*. Springer-Verlag, Rennes, France, 7–20.
- [4] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 years Later: a Literature Review. *Information Systems* 35, 6 (2010), 615–636.
- [5] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2010. Variability modeling in the real: a perspective from the operating systems domain. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, IEEE, Lawrence, KS, USA, 73–82.
- [6] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering* 39, 12 (Dec 2013), 1611–1640.
- [7] Armin Biere, Marijn J.H. Heule, Hans van Maaren, Toby, and Walsh. 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, Amsterdam, The Netherlands, The Netherlands, 697–698 pages.
- [8] Pim Van Den Broek and Ismēnia Galvão. 2009. Analysis of Feature Models using Generalised Feature Trees. In *Third International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain, January 28–30, 2009. Proceedings (ICB Research Report)*, David Benavides, Andreas Metzger, and Ulrich W. Eisenecker (Eds.). Vol. 29. Universität Duisburg-Essen, 29–35. [http://www.vamos-workshop.net/proceedings/VaMoS\\_2009\\_Proceedings.pdf](http://www.vamos-workshop.net/proceedings/VaMoS_2009_Proceedings.pdf)
- [9] Randal E. Bryant. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.* 8, C-35 (1986), 677–691.
- [10] Krzysztof Czarnecki and Peter Chang. 2005. Cardinality-based feature modeling and constraints: A progress report. In *International Workshop on Software Factories*. ACM, 16–20.
- [11] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the Kconfig Semantics and Its Analysis Tools. In *Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2015)*. ACM, New York, NY, USA, 45–54. <https://doi.org/10.1145/2814204.2814222>
- [12] Abdelrahman Osman Elfaki, Somnuk Phon-Amnuaisuk, and Chin Kuan Ho. 2008. Knowledge Based Method to Validate Feature Models. In *12th Software Product Line Conference (SPLC)*, Vol. 2. IEEE Computer Society, Los Alamitos, CA, USA, 217–225.
- [13] Abdelrahman Osman Elfaki, Somnuk Phon-Amnuaisuk, and Chin Kuan Ho. 2009. Using First Order Logic to Validate Feature Model. In *Third International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain, January 28–30, 2009. Proceedings (ICB Research Report)*, David Benavides, Andreas Metzger, and Ulrich W. Eisenecker (Eds.), Vol. 29. Universität Duisburg-Essen, 169–172. [http://www.vamos-workshop.net/proceedings/VaMoS\\_2009\\_Proceedings.pdf](http://www.vamos-workshop.net/proceedings/VaMoS_2009_Proceedings.pdf)
- [14] David Fernandez-Amoros, Ruben Heradio, Carlos Cerrada, Enrique Herrera-Viedma, and J Cobo Manuel. 2017. Towards Taming Variability Models in the Wild. In *New Trends in Intelligent Software Methodologies, Tools and Techniques: Proceedings of the 16th International Conference SoMeT\_17*, Vol. 297. IOS Press, Amsterdam, The Netherlands, 454.
- [15] David Fernandez-Amoros, Ruben Heradio, and Jose Antonio Cerrada. 2009. Inferring Information from Feature Diagrams to Product Line Economic Models. In *Proceedings of the 13th International Conference on Software Product Lines*. Carnegie Mellon University, Pittsburgh, PA, USA, 41–50.
- [16] Rohit Gheyi, Tiago Massoni, and Paulo Borba. 2006. A theory for feature models in alloy. In *Proceedings of the ACM SIGSOFT First Alloy Workshop*. ACM, 71–80.
- [17] Stefan Hengelein and Daniel Lohmann. 2015. *Analyzing the Internal Consistency of the Linux KConfig Model*. Master's thesis, University of Erlangen, Dept. of Computer Science, 2015.
- [18] Ruben Heradio, David Fernandez-Amoros, Jose A. Cerrada, and Ismael Abad. 2013. A Literature Review on Feature Diagram Product Counting and Its Usage in Software Product Line Economic Models. *International Journal of Software Engineering and Knowledge Engineering* 23, 08 (2013), 1177–1204.
- [19] Ruben Heradio, Hector Perez-Morago, David Fernandez-Amoros, Francisco Javier Cabrerizo, and Enrique Herrera-Viedma. 2016. A bibliometric analysis of 20 years of research on software product lines. *Information and Software Technology* 72 (2016), 1 – 15.
- [20] Kyo Kang, Sholom Cohen, James Hess, William Novak, and Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Software Engineering Institute.
- [21] Christian Kästner. 2017. Differential Testing for Variational Analyses: Experience from Developing KConfigReader. *CoRR* abs/1706.09357 (2017). arXiv:1706.09357
- [22] Donald Knuth. 2009. *The Art of Computer Programming, Volume 4, Bitwise Tricks & Techniques; Binary Decision Diagrams*. Pearson Education, Reading, Massachusetts.
- [23] Mike Mannion. 2002. Using First-Order Logic for Product Line Model Validation. In *2nd International Conference on Software Product Lines*. Springer-Verlag, London, UK, 176–187.
- [24] Mike Mannion and Javier Camara. 2004. Theorem proving for product line model verification. *Lecture Notes in Computer Science* 3014 (2004), 211–224. [https://doi.org/10.1007/978-3-540-24667-1\\_16](https://doi.org/10.1007/978-3-540-24667-1_16)
- [25] Marcilio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*. Carnegie Mellon University, 231–240.
- [26] Camille Salinesi, Colette Roll, and Raúl Mazo. 2009. Vmware: Tool support for automatic verification of structural and semantic correctness in product line models. (2009), 173–176 pages. [http://www.vamos-workshop.net/proceedings/VaMoS\\_2009\\_Proceedings.pdf](http://www.vamos-workshop.net/proceedings/VaMoS_2009_Proceedings.pdf)
- [27] Steven She. 2008. *Feature model mining*. Master's thesis, University of Waterloo.
- [28] Steven She. 2013. *Feature model synthesis*. Ph.D. Dissertation, University of Waterloo.
- [29] Steven She and Thorsten Berger. 2010. *Formal semantics of the Kconfig language*. Technical Report, University of Waterloo.
- [30] Jing Sun and Yuan Fang Li. 2005. *Formal Semantics and Verification for Feature Modeling*. Technical Report.
- [31] Reinhard Tartler. 2013. *Mastering Variability Challenges in Linux and Related Highly-Configurable System Software*. Ph.D. Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg.
- [32] Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 531–551.
- [33] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. 2008. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software* 81, 6 (Jun 2008), 883–896. <https://doi.org/10.1016/j.jss.2007.10.030>
- [34] P. Trinidad, D. Benavides, and A. Ruiz-Cortés. 2006. A first step detecting inconsistencies in feature models. In *CAiSE Short Paper Proceedings*.
- [35] G. S. Tseitin. 1983. *On the Complexity of Derivation in Propositional Calculus*. Springer Berlin Heidelberg, Berlin, Heidelberg, 466–483.
- [36] Thomas von der Massen and Horst Licher. 2004. RequiLine: A Requirements Engineering Tool for Software Product Lines. *Lecture Notes in Computer Science* 3014 (2004), 168–180. [https://doi.org/10.1007/978-3-540-24667-1\\_13](https://doi.org/10.1007/978-3-540-24667-1_13)
- [37] Martin Walch, Rouven Walter, and Wolfgang Küchlin. 2015. Formal Analysis of the Linux Kernel Configuration with SAT Solving. In *17th International Configuration Workshop*. University of Helsinki, Helsinki, Finland, 131–137.
- [38] Hai Wang, Yuan Fang Li, Jing Sun, Hongyu Zhang, and Jeff Pan. 2005. A semantic web approach to feature modeling and verification. In *Workshop on Semantic Web Enabled Software Engineering*, 44.
- [39] Haiyan Zhao Wei Zhang, Hua Yan and Zhi Jin. 2008. A bdd-based approach to verifying clone-enabled feature models' constraints and customization. *Lecture Notes in Computer Science* 5030 (2008), 186–199.
- [40] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés. 2008. Automated Diagnosis of Product-Line Configuration Errors in Feature Models. In *Proc. 12th Int. Software Product Line Conf.* IEEE, 225–234. <https://doi.org/10.1109/SPLC.2008.16>
- [41] Hua Yan, Wei Zhang, Haiyan Zhao, and Hong Mei. 2009. An Optimization Strategy to Feature Models' Verification by Eliminating Verification-Irrelevant Features and Constraints. In *Formal Foundations of Reuse and Domain Engineering*, Stephen H. Edwards and Gregory Kulczycki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 65–75.
- [42] Lamia Abo Zaid, Frederic Kleinermann, and Olga De Troyer. 2009. Applying semantic web technology to feature modeling. In *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 1252–1256.
- [43] Christoph Zengler and Wolfgang Küchlin. 2010. Encoding the Linux kernel configuration in propositional logic. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration*, Vol. 2010. IOS Press, Amsterdam, The Netherlands, 51–56.
- [44] Wei Zhang, Hong Mei, and Haiyan Zhao. 2006. Feature-driven requirement dependency analysis and high-level software design. *Requirements Engineering* 11 (2006), 205–220. <https://doi.org/10.1007/s00766-006-0033-x>
- [45] Wei Zhang, Haiyan Zhao, and Hong Mei. 2004. A propositional logic-based method for verification of feature models. In *International Conference on Formal Methods and Software Engineering*. Springer, 115–130.

# Using Relation Graphs for Improved Understanding of Feature Models in Software Product Lines

Slawomir Duszynski

Robert Bosch GmbH

71701 Schwieberdingen, Germany

[slawomir.duszynski@bosch.com](mailto:slawomir.duszynski@bosch.com)

Saura Jyoti Dhar

Robert Bosch GmbH

71701 Schwieberdingen, Germany

[saura.jyoti@bosch.com](mailto:saura.jyoti@bosch.com)

Tobias Beichter

Robert Bosch GmbH

71701 Schwieberdingen, Germany

[tobias.beichter@bosch.com](mailto:tobias.beichter@bosch.com)

## ABSTRACT

Feature models are widely used for describing the variability of a software product line. A feature model contains a tree of features and a set of constraints over these features, which define valid feature combinations. In the industrial practice, large feature models containing hundreds of features and constraints are common. Furthermore, in a hierarchical product line a feature model can be related to other feature models through inter-model constraints. Due to the model size and complexity, understanding industrial feature models is a challenging task.

In this paper, we describe the feature model understanding challenges reported by feature model developers at Robert Bosch GmbH. To support the developers in model understanding, we extend the idea of a feature implication graph to feature relation graph by abstracting groups of implications to feature relations. A transitively closed relation graph shows all modeled and implicit feature relations and spans all related feature models. The graph is also used to identify modeling problems, such as false optional or dead features, and to show the derivation of any implicit relation or problem from the modeled constraints. In a case study at Bosch, we evaluate the use of feature relation graph for model understanding. We propose further use cases of the graph, supporting model maintenance, evolution and configuration.

## CCS CONCEPTS

• Software and its engineering → Software product lines;

## KEYWORDS

Feature model, implication graph, model understanding

### ACM Reference format:

Slawomir Duszynski, Saura Jyoti Dhar and Tobias Beichter. 2019. Using Relation Graphs for Improved Understanding of Feature Models in Software Product Lines. In *Proceedings of the 23rd International SPL conference (SPLC'19)*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3336294.3336317>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org

*SPLC'19, September 9–13, 2019, Paris, France*

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3336317>

## 1 Introduction

A feature model describes the variability of a software product line (SPL) by defining *features* that may vary between product variants – such as their different functionalities, non-functional characteristics, and other attributes. The feature model also defines *constraints* on the features, expressed in the form of the feature tree structure (e.g., parent-child relations, alternatives), as cross-tree constraints (e.g., *A requires B*), or as propositional formulas [6][9]. A feature model of a large SPL can contain hundreds or even thousands of features and constraints [3][21]. Furthermore, a feature model can be decomposed into several related fragments. In a hierarchical SPL, the product line consists of components which are themselves variable, and can have their own feature models [12]. Similarly, a decomposition of a large feature model according to the detail level or development phase is practiced [19]. The resulting partial feature models are related to each other by inter-model constraints, that is, constraints defined on features from two or more models. Hence, the group of related models represents logically a single large feature model, while being physically divided into several model files.

**Problem.** Understanding large feature models, fragmented into many files, is a challenging task. We motivate this in Section 2 based on the case study in this paper, which is a mature SPL of engine control unit (ECU) software developed by the PS-EC department of Robert Bosch GmbH [7][20]. Along with the size of the models, we identified that the difficulties in model understanding result from the large number of *implicit* constraints between the features. For example, the *explicit* (i.e., modeled) constraints  $A \Rightarrow B$  and  $B \Rightarrow C$  result in an *implicit* constraint  $A \Rightarrow C$ . In a model rich in explicit constraints, implicit constraints resulting from a conjunction of ten or more explicit constraints can occur (as shown in Section 5). Such implicit constraints are very difficult to detect manually, even more if the chain of linked constraints crosses the boundary of a single feature model. Furthermore, if the implicit constraint is known to exist, it is still difficult to find its reason, i.e., to determine the corresponding chain of explicit constraints. Finally, assuring an absence of an implicit constraint is in this situation also challenging. Note that while especially the last question can be answered using a SAT solver, the developers typically have no access to a solver and no knowledge how to use it. Instead, they rely on their understanding of the models, the constraints within them, and the domain.

**Contribution.** To support understanding of complex feature models, we define feature relation graphs. The relation graphs extend the idea of feature implication graphs proposed by Czarnecki et al. [6] by grouping implication subgraphs into abstracted, higher-level feature relations. We calculate the transitive closure of the relation graph, obtaining for every feature its implicit constraints, including the constraints that refer to features of other models and hence cross the model boundary. Additionally, the relation graph identifies modeling problems such as dead features. For every implicit constraint or problem, its complete derivation from the explicit constraints is provided to the user, supporting analysis. We extend the pure::variants feature modeling tool [4] with a graph view showing the constraints, the modeling problems, and the derivations. In a case study, the feature model developers at Robert Bosch GmbH use the approach and report improved model understanding.

The contributions of this paper are therefore:

- We report the practical challenges in understanding complex feature models, experienced by developers of a large, hierarchical industrial SPL.
- Based on the feature implication graph approach and on the concept of feature relations used in pure::variants, we define modal feature relation graphs. The graphs represent explicit and implicit model constraints, with non-binary constraints stored as modal edges. The graphs enable finding modeling problems such as dead features, provide explanation of implicit constraint derivation, and span many related feature models.
- We report an application of the approach to example industrial models. We describe a case study of using the relation graphs to improve feature model understanding.
- We discuss further ideas of using the feature relation graphs for feature modeling and configuration.

**Paper structure.** In Section 2, we report the practical challenges in understanding complex feature models. We also introduce an industrial feature model and a running example, which are both used in later sections to illustrate the approach. In Section 3, we present the related work: the original idea of feature implication graphs, and the feature relation concept of pure::variants. In Section 4, we describe our approach of feature relation graphs, and motivate how it supports understanding of complex feature models. In Section 5 we apply the approach to the example feature models. In Section 6 we report the case study results. In Section 7 we discuss the benefits and drawbacks of the relation graph approach. In Section 8 we describe further possibilities of using the feature relation graph for supporting the feature modeling and configuration tasks. Section 9 concludes the paper.

## 2 Challenges in Understanding Feature Models

**Context.** Our case study is the engine control unit SPL developed by Robert Bosch GmbH. The SPL consists of about 300 independently managed components, and delivers about 2000 different products per year. Detailed feature models, created in pure::variants, exist for some components. Additionally, a system-level feature model captures the SPL variability on an abstract level. Component feature models frequently have more than 100 features and more than 100 internal constraints. In addition, the component feature models refer to the main system feature model using inter-model constraints.

**Example Bosch models.** Figure 1 depicts a group of feature models maintained by a Bosch development team responsible for two example closely related engine control components, A and B. Each of these components has its own feature model with internal constraints. The components also contain inter-model constraints related to the features of the main system model, depicted as arrows in Figure 1. In this example, only a sub-tree of the main model related to components A and B is shown. The main feature model is not allowed to contain inter-model constraints for maintainability reasons. In addition, explicit relations between component features are discouraged (only 2 such constraints exist in the example). However, many implicit relations between features of component A and component B exist, as inter-model constraints in both component feature models can refer to the same features of the main model. In total, the example models contain 360 features and 273 hard constraints. The tree structure of the model also expresses further constraints, e.g., in the form of parent-child relations and alternative feature groups.

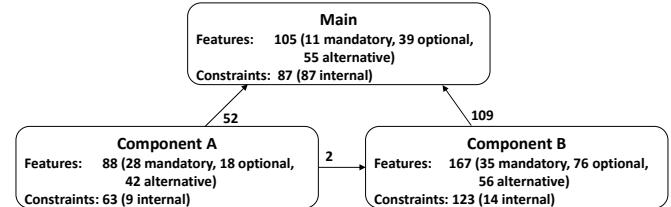
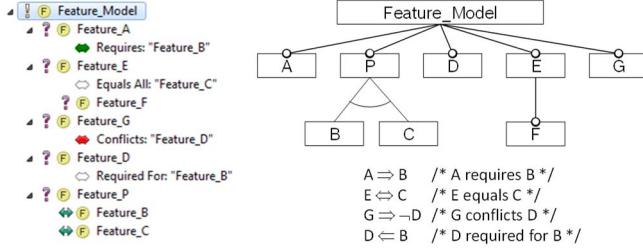


Figure 1: The feature models of the case study components

**Model understanding challenges.** At Bosch, the feature model developers are the most experienced members of the component team, and have deep domain knowledge. Despite their knowledge and experience however, the model developers of these as well as other models in the engine control SPL report difficulties in answering the following types of questions about their models:

- **Q1. Determining feature selection effect.** When feature A is selected, what are all resulting selections and deselections of other features?
- **Q2. Selection effect explanation.** Why does feature F become deselected after selecting feature A, although there is no constraint relating F and A, and no other feature is related to both F and A?
- **Q3. Preserving model correctness.** Can a new constraint relating features A and G create a conflict in the model? For example, can it create a dead feature? If the new constraint is correct, which of the previously existing constraints created the problem?
- **Q4. Following inter-model constraints.** In which way do the inter-model constraints defined in model M2 result in implicit constraints between features of model M1? For example, can two features from model M1 be mutually exclusive, although there is no explicit nor implicit constraint between them within the scope of model M1?
- **Q5. Analyzing the impact of a model change.** When adding or removing an explicit constraint, what are the resulting changes in the implicit constraints in the whole model or model group?

The above challenges are illustrated in the following two figures. Figure 2 depicts a simple feature model, which is a second running example in this paper. The model is shown in both pure::variants and FODA [9] notations, as the Bosch developers use the pure::variants tool for feature modeling. The pure::variants modeling concepts are equivalent to FODA and explained in Section 3. To illustrate the model understanding challenges, please consider the explicit and implicit constraints of feature A in Figure 2. We encourage now the reader to use this example to answer the first three question types (Q1–Q3) from the above challenge list before proceeding to their explanations described below.

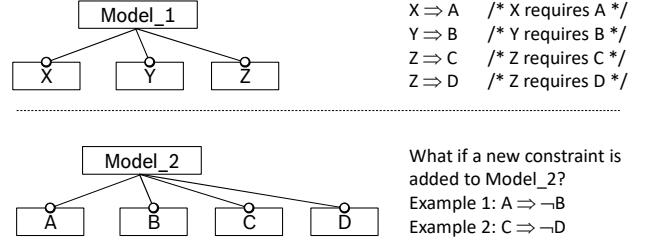


**Figure 2: A running example of a feature model shown in the pure::variants (left) and FODA (right) notations**

**Example explanation.** A careful analysis of the above model shows that feature A is involved in a constraint, explicit or implicit, with every other feature. The constraint  $A \Rightarrow B$  is explicitly modeled. Implicitly, since  $B \Rightarrow D$  and  $B \Rightarrow P$ , also  $A \Rightarrow D$  and  $A \Rightarrow P$ . Due to the alternative constraint between B and C, we have  $A \Rightarrow \neg C$ . Also, since  $E \Leftrightarrow C$  and F is a child of E, we have  $A \Rightarrow \neg E$  and  $A \Rightarrow \neg F$ . Finally,  $G \Rightarrow \neg D$  means also that  $D \Rightarrow \neg G$ , so due to  $A \Rightarrow D$  we have  $A \Rightarrow \neg G$ . Consequently, a selection of A results in selections of B (due to 1 explicit constraint), D (2 constraints), and P (2), and in deselections of C (2), E (3), F (4) and G (3 explicit constraints). Adding a new constraint to the example model is either redundant, or likely to create a dead feature. For example, adding a constraint  $A \Rightarrow G$  would make A to a dead feature, because since  $A \Rightarrow \neg G$ , there would be no valid configuration including A. In case the constraint  $A \Rightarrow G$  is correct and should be included, the modeler would need to correct the model by understanding which of the three already existing explicit constraints  $A \Rightarrow B$ ,  $B \Rightarrow D$  and  $D \Rightarrow \neg G$  needs to be removed.

Note that in the practical case the group of related explicit constraints might be spread over several subtrees of a large feature model. In that situation, keeping an overview of the constraints is much more difficult than in Figure 2 and the model understanding challenges are amplified.

**Inter-model constraints.** Figure 3 depicts two related feature models, Model 1 and Model 2, which are maintained by different developers. Model 2 is edited and configured independently, while Model 1 depends on Model 2 as it defines inter-model constraints. To illustrate the understanding challenges Q4 (finding implicit constraints created by explicit inter-model constraints) and Q5 (assessing model change impact), consider two constraint additions:



**Figure 3: Example problems due to inter-model constraints**

- Adding a new constraint  $A \Rightarrow \neg B$  to Model 2. This causes the features X and Y of Model 1 to become mutually exclusive.
- Adding a new constraint  $C \Rightarrow \neg D$  to Model 2. This results in the feature Z of Model 1 becoming a dead feature.

In both cases, the new constraints are added to Model 2 and do not cause any problems in the scope of that model. It is easy for the modeler of Model 2 to overlook the influence of these constraints on Model 1. At the same time, the content of Model 1 did not change, but now that model contains a new implicit constraint, which is not easy to find. For developer of Model 1, one way of finding the new constraint would be to review the changes done to Model 2. This is however not practical if Model 2 is large and changes frequently. Another way to detect the constraint would be to observe its effects when creating a configuration of Model 1. However, this might also be easily overlooked if Model 1 is large and the features bound by the implicit constraint are located in different sections of the feature tree. In the Bosch PS-EC context, Model 2 corresponds to the system-level feature model while Model 1 is a component feature model.

**Dead and always-selected features.** Dead features are features that cannot be selected in any valid configuration. In the literature, dead and always-selected features are considered a modeling flaw [2]. However, in the Bosch PS-EC feature models there are legacy features that are consciously modeled as dead or always selected. The reason for this is the life cycle of a feature. After a new feature is introduced, it is frequently only used in specific or high-end products. After a few years however, the feature either becomes a standard element of all products, or becomes obsolete due to new technology. Hence, such a legacy feature does not represent active variability anymore. A consistent removal of the legacy feature from all relevant variation point conditions in the assets is a refactoring activity, which requires time. Consequently, the feature model needs to hold such features until refactoring completes, and specifies them as dead or always selected to ensure their correct configuration.

To account for this modeling practice, we integrate the detection and presentation of modeling flaws, including dead and always selected features, in the feature graph view. This enables the developers to confirm the existence of the dead features they intended to model, as well as to find the unintended ones.

**Feature implication graphs.** Our approach addressing the above model understanding challenges bases on the idea of feature implication graphs. We introduce the origins of the idea in Section 3 as related work, and then we describe our approach in Section 4.

### 3 Background and Related Work

Our approach extends previous work on implication graphs done by other authors. Consequently, we include here the details of related work that are later needed to explain our approach.

**Feature model as implication graph.** Czarnecki et al. [6] first introduced the idea of representing a feature model as a directed implication graph. In the implication graph, two nodes are created for every feature – one representing the affirmation of the feature, and one representing its negation. Additionally, two further nodes representing the Boolean false and true literals are created. Every edge of the graph represents an implication, expressing the binary dependencies of the model: the A requires B constraint as  $A \Rightarrow B$ , the A conflicts B constraint as  $A \Rightarrow \neg B$ , an optional child tree relation as  $\text{child} \Rightarrow \text{parent}$ , and a mandatory child relation as two implications  $\text{child} \Rightarrow \text{parent}$  and  $\text{parent} \Rightarrow \text{child}$ . Constraints that are more complex are transformed to non-binary implications (having many premises) and expressed as hyperedges of the graph (i.e., edges connecting more than two nodes). For example, an or-group of two features *child1* and *child2* located under a feature *parent* can be expressed as implications  $\text{parent} \wedge \neg \text{child1} \Rightarrow \text{child2}$  and  $\text{parent} \wedge \neg \text{child2} \Rightarrow \text{child1}$ . An alternative group is expressed as an or-group with an additional conflict constraint between the children, i.e.,  $\text{child1} \Rightarrow \neg \text{child2}$ .

**Example graph.** Figure 4 depicts the implication graph constructed for the feature model from Figure 2. Note that for a single binary implication from the original model, exactly two edges are created in the implication graph. This results from the application of logic laws. For any two features  $f_1, f_2$ :

- If an implication  $f_1 \Rightarrow f_2$  is added to the graph, its logical equivalent  $\neg f_2 \Rightarrow \neg f_1$  also needs to be added and vice versa. The statement  $(f_1 \Rightarrow f_2) \Leftrightarrow (\neg f_2 \Rightarrow \neg f_1)$  is a tautology.
- Similarly, if an implication  $f_1 \Rightarrow \neg f_2$  is added, also its equivalent  $f_2 \Rightarrow \neg f_1$  is added, because  $(f_1 \Rightarrow \neg f_2) \Leftrightarrow (f_2 \Rightarrow \neg f_1)$ .
- Analogically, for  $\neg f_1 \Rightarrow f_2$  also  $\neg f_2 \Rightarrow f_1$  is added, because  $(\neg f_1 \Rightarrow f_2) \Leftrightarrow (\neg f_2 \Rightarrow f_1)$ .

Furthermore, the root of the example feature model is a mandatory feature and is hence always selected. We use this fact to simplify the graph in Figure 4 by merging the nodes TRUE and FEATURE\_MODEL, and merging their negations.

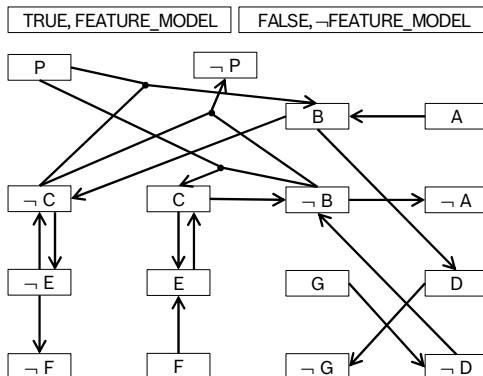


Figure 4: The example feature model represented as implication hypergraph

In Figure 4, we also omit for any feature  $f$  the trivial implications  $f \Rightarrow \text{TRUE}$ ,  $\neg f \Rightarrow \text{TRUE}$ ,  $\text{FALSE} \Rightarrow f$  and  $\text{FALSE} \Rightarrow \neg f$ .

**Completeness of implication graphs.** An implication graph with no hyperedges can express any binary relation between two features (as shown in Figure 6 in Section 4). However, for non-binary constraints the hyperedges are necessary, and complicate the graph handling. Knüppel et al. [10] investigated the use of non-binary constraints in industrial and open source feature models. They differentiated between pseudo-complex constraints, which can be logically decomposed into binary constraints, and strict-complex constraints for which this is not possible. In the investigated models, between 1% and 74% of constraints were strict-complex, which indicates that using binary relations is not sufficient for the more complex models. To counteract this problem, they developed an algorithm transforming the strict-complex constraints into binary constraints through the introduction of additional features to the original model. Hence, binary implication graphs can also be constructed for the complex feature models after the model transformation is applied.

An alternative approach ensuring the completeness of the graph is presented by Krieter et al. [11]. Instead of using hyperedges, they introduce a modal implication graph having strong and weak edges, i.e., binary graph edges that are unconditional or have a presence condition. Hence, a single non-binary constraint is decomposed into multiple weak edges. Since we use a similar approach to represent non-binary constraints in our graphs, we illustrate the modal implication graph idea using the example feature model in Figure 5.

**Uses of the implication graph.** An implication graph has several applications. For example, Czarnecki et al. [6] use the implication graph as an alternative representation of a feature model for the purpose of model analysis. They observe that the transitive closure of the graph as well as its transitive reduction are logically equivalent to the input feature model. She et al. [18] use the graph as an intermediate representation for reverse engineering feature models of open source operating systems. Krieter et al. [11] use a modal implication graph for configuration decision propagation, using strong and weak graph edges. Strong edges are used directly for decision propagation, while the weak edge conditions become true (activating the edge) or false (removing the edge) as the features are configured.

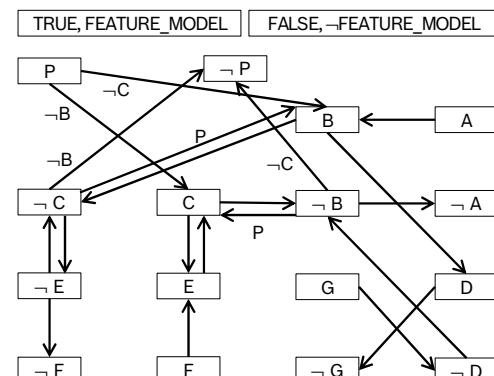


Figure 5: The example feature model represented as modal implication graph

**Implication graphs for feature model understanding.** An implication graph can be used as a visualization of model constraints. For example, answering the model understanding questions from Section 2 can be supported by Figure 4 or Figure 5. However, viewing the complete graph and following the paths of implication edges is not practical for larger models due to the number of nodes and edges. Instead, a local view from the perspective of a given feature can be used. Krieter et al. [11] use a transitive closure of the graph to be able to quickly determine for each node all other nodes reachable by a directed path of constraints. While they do this due to performance reasons, the same idea can be applied for a human viewing the model. Hence, to find the constraints of a given feature, it is enough to look at the incoming and outgoing edges of the respective positive and negative node in the transitively closed implication graph. Martinez et al. [13] use this idea and present a view showing all related graph nodes for a single node of the transitively closed implication graph. The purpose of the view is to aid constraint definition based on implications that are reverse engineered from product configurations. The view presents both hard and soft constraints (i.e., constraints that only hold for a subset of configurations). However, at a given time the view can only show one feature node (i.e., only the positive or only the negative one) and one of the two possible implication directions (node to other features, or other features to node). Hence, analyzing four views is necessary for a complete overview of feature relations. Martinez et al. state that the use of four views was confusing for the users, and they hence only used one view (i.e., only a part of the implication graph information) for constraint discovery. In contrast to that, in Section 4 we describe an approach using the complete graph information in one view, and abstracting the relation types between features for easier understanding. The pure::variants tool [4][16] supports two forms of constraint modeling: textual logical expressions in the pvSCL language, and feature relations. The use of feature relations is preferred in the tool for understandability reasons. The relations resemble the implication hypergraph as they define implications with one-to-many premises, one-to-many conclusions, and the use of affirmed or negated features. However, the use of relations abstracts and hides the actual implications from the user and instead displays the relations as typed hyperedges between the features. A relation has one source S, and can have one or many targets T<sub>1</sub> ... T<sub>n</sub>. Table 1 lists the most important pure::variants relation types.

pure::variants relation	Logical equivalent
S requires T <sub>1</sub> (or T <sub>2</sub> (...))	S $\Rightarrow$ (T <sub>1</sub> or ... or T <sub>n</sub> )
S requiresAll T <sub>1</sub> (and T <sub>2</sub> (...))	S $\Rightarrow$ (T <sub>1</sub> and ... and T <sub>n</sub> )
S requiredFor T <sub>1</sub> (or T <sub>2</sub> (...))	(T <sub>1</sub> or ... or T <sub>n</sub> ) $\Rightarrow$ S
S requiredForAll T <sub>1</sub> (and T <sub>2</sub> (...))	(T <sub>1</sub> and ... and T <sub>n</sub> ) $\Rightarrow$ S
S conflicts T <sub>1</sub> (and T <sub>2</sub> (...))	(T <sub>1</sub> and ... and T <sub>n</sub> ) $\Rightarrow$ $\neg$ S
S conflictsAny T <sub>1</sub> (or T <sub>2</sub> (...))	(T <sub>1</sub> or ... or T <sub>n</sub> ) $\Rightarrow$ $\neg$ S
S equalsAny T <sub>1</sub> (or T <sub>2</sub> (...))	S $\Leftrightarrow$ (T <sub>1</sub> or ... or T <sub>n</sub> )
S equalsAll T <sub>1</sub> (and T <sub>2</sub> (...))	S $\Leftrightarrow$ (T <sub>1</sub> and ... and T <sub>n</sub> )

Table 1: The most important pure::variants relation types

A single pure::variants relation subsumes and abstracts several implications of the implication graph. The use of different relation types, accompanied by color-coding the relations based on the type, helps the modeler understand the given logical constraint. We use the idea of relation types in our approach described in Section 4. Furthermore, pure::variants provides a relation view for a selected feature, showing all relations in which the feature is the source or target. The view integrates all relation types and also shows inter-model relations. However, the view only shows the explicitly modeled relations, the implicit ones are not presented – leading to the model understanding problems described in Section 2.

**Feature model understandability.** Botterweck et al. [5] address the challenge of explaining feature dependencies (Q1) by using transitive closure of requires, excludes and implements relations. However, they do not address the understanding challenges Q2-Q5 and do not consider all possible Boolean constraints.

Urli et al. [23] support understanding constraints in a group of feature models by visualizing the count of internal and inter-model constraints in model subtrees. Jakšić et al. [8] experimentally find that a visual feature modeling notation supports human understanding better than a textual one. Both these works relate to a similar problem as our paper, but do not address the implicit constraints. Acher et al. [1] and Urli et al. [22] investigate constraints in groups of related feature models. As in our case, they observe that constraints propagate between the models, and adding one model can restrict the configurability of another one. They propose approaches for consistent composition and configuration of already defined models. In contrast to that, we target the consistency and model quality during the earlier model definition phase.

Several authors, for example Pohl et al. [15], address feature model complexity by defining respective metrics. While this contributes to characterization of the models, the metrics we are aware of do not address the problem of model constraint understanding.

## 4 The Feature Relation Graph

**Basic ideas.** Inspired by the pure::variants concept of relations, we extend the idea of feature implication graph to a feature relation graph:

- **Relations instead of implications.** A feature relation graph represents the feature model as a graph, in which constraints are represented as binary relation edges. A single relation abstracts and subsumes several implications, as discussed in Section 3. This reduces the number of graph edges.
- **Single feature nodes.** The use of relations makes it possible to represent each feature by just one node. Additionally, the graph contains a node representing the Boolean literal TRUE.
- **Modal relation graph.** All edges in the feature relation graph are binary. To represent constraints that cannot be reduced to a binary relation form, weak relation edges having a presence condition are used.
- **Transitive closure for understanding.** To support identification of implicit constraints in the feature model, the transitive closure of the relation graph is calculated. Hence, for a given feature all other features related to it can be identified by viewing the incoming and outgoing relation edges of the feature node.

- Relation explanation.** When calculating the transitive graph closure, the newly added relation edges representing implicit constraints are built based on the explicit constraint edges and on the previously found implicit constraint edges. Each new implicit edge stores the identifiers of the edges that contributed to its creation. This information is presented to the user on demand to support understanding the implicit constraints.

In the remainder of this section, we discuss in detail the realization of these concepts.

**Replacing implications by relations.** As discussed before, binary implications in the original feature implication graph always form pairs. This is due to the logical equivalence of the following implications for any two features A, B:

- (1)  $(A \Rightarrow B)$  is equivalent to  $(\neg B \Rightarrow \neg A)$ .
- (2) Symmetrically,  $(B \Rightarrow A)$  is equivalent to  $(\neg A \Rightarrow \neg B)$ .
- (3)  $(A \Rightarrow \neg B)$  is equivalent to  $(B \Rightarrow \neg A)$ .
- (4)  $(\neg A \Rightarrow B)$  is equivalent to  $(\neg B \Rightarrow A)$ .

We use the four above implication pairs to define the basic relation types in the feature relation graph. Specifically, we define the pair (1) as A *requires* B, the pair (2) as A *required\_for* B, the pair (3) as A *excludes* B, and the pair (4) as A or B.

These four relation types are also depicted in the first row of Figure 6. The figure consists of 16 parts showing all 16 possible binary functions of two logical variables. In the top section of every part, the relation name and the notation of the logical function are given. The middle section shows the implication graph representing the function. The bottom section contains the truth table of the function. The feature model is only satisfied if the function has the value of true. To help identify the functions, we assign them unique numbers in the 0 to 15 range derived from the binary coding of the function values. For example, the relation A *excludes* B, i.e., the function  $\neg(A \wedge B)$ , is assigned the number 7. Figure 6 illustrates the following elements of our approach:

- Completeness.** The conjunctions of the 4 basic relation types are sufficient to express all 16 binary functions. For every function, the figure provides the respective conjunction in an oval below the logical function notation.
- Names for understanding.** For 15 binary functions (all except TRUE), the user is shown the relation name as given in the top section of the respective figure part. In fact, only the names *equals*, *alternative*, *dead*, *always\_selected* and *unsatisfiable* need to be added to the four basic relation names to completely characterize the 15 functions. The relation names enable the user to understand the logical effect of the relation without analyzing the implication graph. The function TRUE does not require any name as it is not displayed as a graph edge.
- Perspective of the selected feature.** The relation view displays the relations from the perspective of the selected feature. Hence, the display name of some relations and the display order of the feature names need to be adapted if the feature selection changes. For example, selecting feature A results in the display of the A *requires* B relation, while after selecting feature B the same relation is shown as

B *required\_for* A. For the symmetric relations such as A *excludes* B only the display order or feature names is changed, here to B *excludes* A.

- On a side note, Figure 6 also shows how the implication graph can represent all 16 logical binary functions of two variables. Consequently, the feature relation graph is a multigraph using four edge types: implication with two possible directions  $A \Rightarrow B$  and  $A \Leftarrow B$ , excludes (NAND), and OR. An edge of a given type can occur at most once between the given two features. For a given feature pair, the presence or absence of the four edge types results in 16 possible logical functions. When displaying the graph for a feature, the name of the respective logical function is shown.

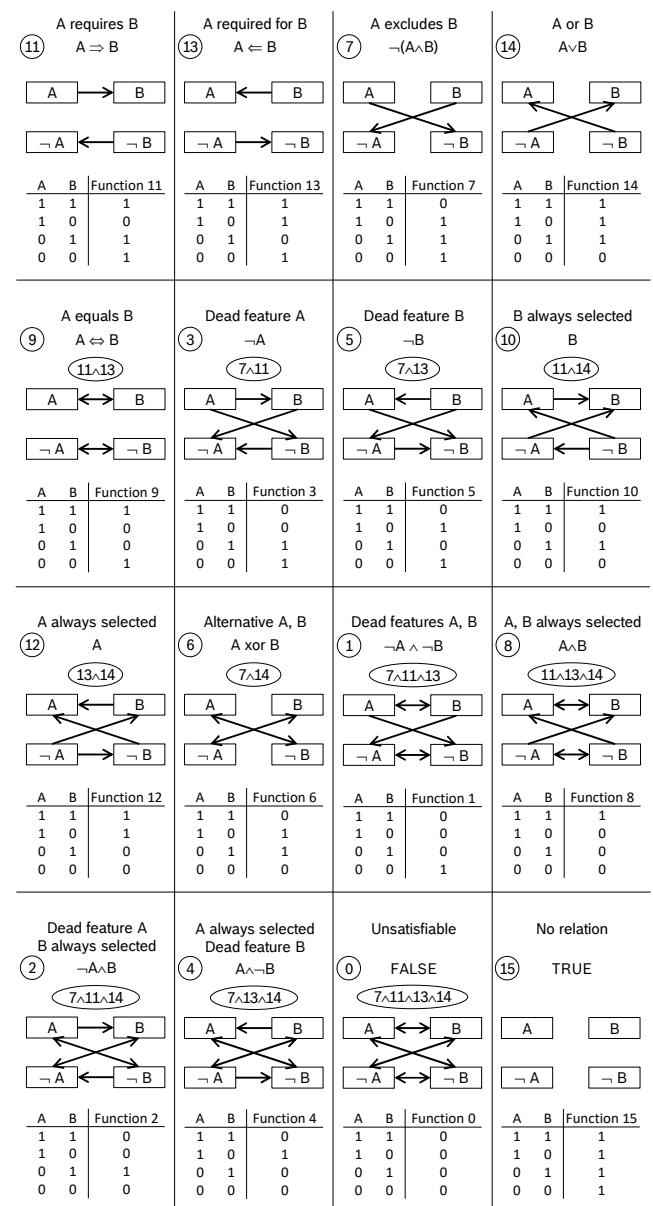
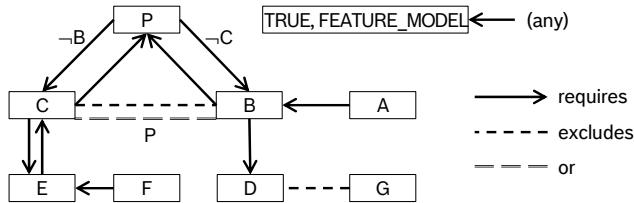


Figure 6: All possible binary functions of two variables, represented as conjunctions of 4 basic relation types. Every function has a name describing its effect.

Finally, note that every relation represents a constraint that needs to be satisfied (i.e., the logical function has to return true) for the feature model to be satisfied and yield a valid product. Hence, the semantics of the *or* and *alternative* relation types is slightly different from the semantics of a binary *or* and *alternative* groups in a feature model. These groups represent a constraint with a presence condition: they need to have at least one (or exactly one) feature selected if and only if their parent is selected. If the parent is deselected in a valid product, the groups have no selected features in that product. An example of this situation is depicted in Figure 7 for features P, B and C. In contrast to that, the *or* and *alternative* relation types need to be satisfied in every valid product and do not depend on the selection of any other feature. This corresponds to a group of two *alternative* or *or* features whose parent is selected in all valid product configurations. Figure 7 shows the example feature model from Figure 2 presented as a feature relation graph. To simplify the figure, the *requires* relations from any feature to the root node are not shown.

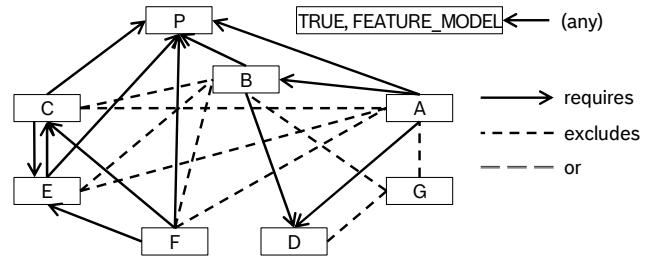


**Figure 7: The example feature model represented as modal feature relation graph**

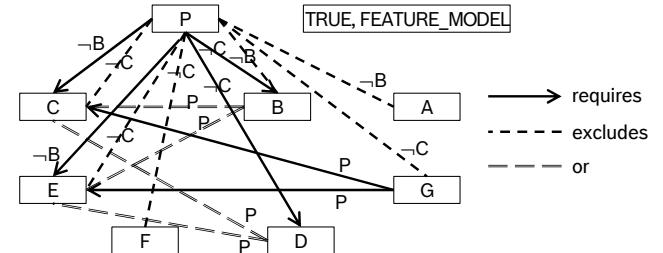
**Modal relation graph.** In Figure 7, three weak edges between the features P, B and C, resulting from the decomposition of the alternative group constraint, are shown. If different strong and weak edges exist between a pair of features, it is possible to conjunct those to conditional binary functions. For example, the strong relation  $B \Rightarrow P$  and the weak relation  $P \Rightarrow B [¬C]$  can result in a “mixed weak” relation  $P \Leftrightarrow B [¬C]$ . In our experience, the developers consider the weak relations more difficult to understand than the strong ones. Hence, in our view we conjunct strong edges to strong relations, and the weak edges are processed and shown separately.

**Transitive graph closure.** The transitive graph closure is calculated for all edge types, resulting in the creation of new strong and weak edges for the implicit constraints. Figure 8 and Figure 9 show respectively the strong and the weak edges of the resulting transitively closed feature relation graph.

Based on the edges of a user-selected feature, the relation graph view shows a table with the relations of that feature. Several filtering options are available and can be combined. For example, the user can choose to see only strong or only weak edges, or only relations to features of its own model or of other models. Also, relations which are purely tree-based (e.g. child $\Rightarrow$ parent, any\_feature $\Rightarrow$ root) are by default hidden, as developers indicated that the tree structure is easy to understand and these relations are obvious. Finally, weak edges for which the relation is always satisfied if their presence condition is true, such as P *excludes* B [ $¬B$ ] and P *excludes* C [ $¬C$ ], are not interesting and are also filtered.



**Figure 8: Strong edges of the transitively closed feature relation graph**



**Figure 9: Weak edges of the transitively closed feature relation graph**

Figure 10 depicts a screenshot of the view implementation, showing the strong relations of feature A. As the view is used with pure:variants, we use the specific pure:variants icons and relation names (such as *conflicts* instead of *excludes*). The icon of the root feature FeatureModel is decorated with a small overlay of a blue circle to denote that this relation is purely tree-based.

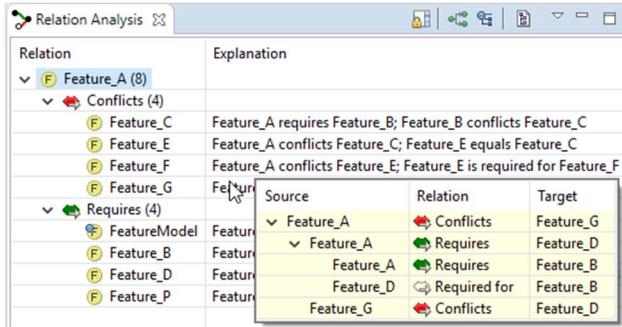
Relation	Explanation
Feature_A (8)	
↳ Conflicts (4)	
Feature_C	Feature_A requires Feature_B; Feature_B conflicts Feature_C
Feature_E	Feature_A conflicts Feature_C; Feature_E equals Feature_C
Feature_F	Feature_A conflicts Feature_E; Feature_E is required for Feature_F
Feature_G	Feature_A requires Feature_D; Feature_G conflicts Feature_D
↳ Requires (4)	
FeatureModel	FeatureModel is a parent of Feature_A
Feature_B	Feature_A requires Feature_B
Feature_D	Feature_A requires Feature_B; Feature_D is required for Feature_B
Feature_P	Feature_A requires Feature_B; Feature_B requires Feature_P

**Figure 10: A screenshot of the relation graph view implementation showing the strong relations of feature A**

The table can also show the complete relation graph, grouping the relations either by their features or by the types. This view can be used to see all dead features or to calculate statistics on the model.

**Explanations.** During the calculation of transitive graph closure, each newly derived graph edge or relation stores pointers to the previously existing edges and relations that contributed to its creation. For example, the relation A *requires* D references the relations A *requires* B and D *requiredFor* B. Similarly, a relation between two features points to its contributing edges, e.g., the edges F1 *requires* F2 and F1 *conflicts* F2 are pointed to by a relation *deadFeature* F1. The transitive closure algorithm runs in multiple iterations – hence, the edges derived in the next iteration might point to edges derived in the previous ones. For example, the edge

A *excludes* G points to G *excludes* D (explicit) and A *requires* D (derived). Consequently, the pointer structure for each derived edge forms a tree, in which the leaf nodes are the explicit edges and the non-leaf edges are derived. An example of such tree is depicted for the A *excludes* G relation in the screenshot in Figure 11.



**Figure 11: A screenshot of the on-demand explanation tree for the derived relation A excludes G**

Using the tree, the modeler can identify all explicit constraints that contribute to the given implicit one. For explicit constraints that form an implication chain, it would also be possible to display just the explicit constraints in the respective order. However, in a general case the final relation can be composed of two or more implicit edges, each of which being a result of a different chain. A tree is more suitable for displaying such complex situations.

**Graph analysis and problem identification.** As depicted in Figure 6, the relation graph can be used to identify dead and always selected features. Additionally, a simple graph analysis is performed to identify and display further facts about the model:

- Features that are modeled as optional children, but are implied by their parent, should be modeled as mandatory children instead. This is indicated in the view with a “false optional” warning and with an explanation of the parent $\Rightarrow$ child relation.
- In the graph, strong and weak relations of the same type can be derived for the same two features. While the derived weak relations are deleted if a strong relation of the same type exists, an explicitly modeled weak relation generates a warning instead. For example, a X *requires* (Y or Z) pure::variants relation results in the explicit weak edges  $X \Rightarrow Y[-Z]$ ,  $X \Rightarrow Z[-Y]$  and  $Y \vee Z [X]$ . If a strong implicit edge  $X \Rightarrow Y$  is identified, the view indicates that the pure::variants relation which created the weak edge can be modified or eliminated, and shows the explanation of the strong  $X \Rightarrow Y$  edge to support the analysis.
- The identified dead and always selected features are replaced by true or false Boolean literals in the weak edge presence conditions. Then, logical simplification of the conditions is run. If a presence condition can be fully evaluated to true or false, the weak edge either becomes a strong one or is deleted. The operation is repeated if the newly identified strong edges result in the identification of new dead or always selected features.

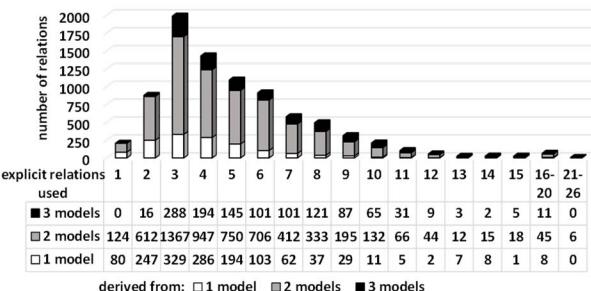
## 5 Example Model Analysis

**Example models.** In Section 2, we introduced three related feature models developed at Bosch PS-EC, having 360 features and 273 constraints. We use now the relation graph to analyze the implicit constraints in these models and illustrate the understanding challenges.

Before calculating the transitive closure, the relation graph built for the three models contains 204 explicit strong relations. The number of strong relations is in this case lower than the number of initial constraints, as some constraints resulted in the creation of weak (conditional) relations, while others were merged to one relation (e.g.,  $A \Rightarrow B$  and  $A \Leftarrow B$  merged to  $A \Leftrightarrow B$ ). Out of the 204 relations, 124 cross the model boundary. Additionally, the graph contains 616 strong relations resulting directly from the tree constraints. Hence, there are 820 strong relations in the initial graph.

**Challenge Q1 (many selection effects).** After calculating the transitive closure, the graph contains 10297 strong relations (including the 820 explicit ones). 1925 relations are based purely on the tree constraints, while 8372 relations are at least partially derived from the modeled constraints. Consequently, each feature participates in many implicit relations: while there are on average 1.13 explicit relations per feature (204 binary strong relations for 360 features), with 8372 transitive non-tree relations there are on average over 46 total relations per feature. Note that we count the relations twice here, as each is relating two features.

**Challenge Q2 (difficult effect explanation).** Figure 12 depicts two metrics for the 8372 non-tree relations: the number of explicit relations used for their creation (horizontal axis) and the number of models where the features used in the explicit relations are defined.



**Figure 12: The number of strong graph relations (vertical axis) built from a given number of explicit relations (horizontal axis) using features from one, two or three example models**

As depicted in the data table of Figure 12, for the majority of the implicit relations (4500) 3 to 5 explicit relations were transitively joined for their creation. The highest number of explicit relations used to create an implicit one is 26, and 298 implicit relations are created from 11 or more explicit ones. The relations having long derivations are particularly difficult for model understanding.

**Challenge Q4 (effects of inter-model constraints).** The most relations (5784) are constructed from explicit relations using features belonging to two models, while 1409 use features of one model only, and 1179 use features of all three models. At the same time, the final binary relations connect 3281 pairs of same-model features and 5091 pairs of different-model features. Consequently, for 3281-1409=1872 relations the both related features are in the same model, but the derivation chain(s) of that relation cross the model boundary going to another model(s) and returning back. Similarly, out of 5091 relations between features of two models, 1171 also use features of the third model in their explanation tree.

**Conclusion.** Based on the above metrics, we conclude that understanding the example group of feature models is a challenging task due to a large number of implicit constraints, long derivation chains, and the derivations crossing the model boundary many times.

## 6 Case Study

**Research questions.** We performed a case study at Robert Bosch GmbH to investigate the following research questions:

- **RQ1.** Do the developers experience the feature model understanding problems related to implicit constraints?
- **RQ2.** Does the use of relation graph reduce the effort and perceived difficulty of feature model understanding tasks?
- **RQ3.** Does the use of relation graph help the developers discover new facts about the constraints in their models?

**Participants.** We asked five feature model developers (three at Bosch PS-EC, two at other Bosch divisions) to participate in the case study. All developers agreed for participation on voluntary basis. All five developers are domain experts and the original creators of their models (one of them created the example models analyzed in Section 5). They received the Eclipse update site for view installation and a document explaining the logic of the relation graph view. We asked them to use the view while performing their actual work tasks using their feature models. With each developer, we performed a pre-interview focusing on RQ1 shortly before providing the view, and a post-interview focusing on RQ2 and RQ3 one to two weeks later.

**Pre-interview (RQ1).** In the pre-interview, all five developers stated that they use the standard pure::variants relation view, which shows the explicitly modeled constraints related to the selected feature. They indicated that the standard view already supports model understanding, as it collects all information related to the explicit usage of a feature in one place. Four developers indicated that, despite using the explicit relation view, they still experience challenges in understanding the implicit relations in their models as described in Section 2. While following a known relation chain is possible, it is time consuming to find all the implicit relations of a feature. Hence, they rated the tasks of explaining a selection effect (challenges Q1, Q2), or ensuring the lack of incorrect selection effects (Q3, Q4), as difficult and effort-intensive.

One PS-EC developer stated that his domain and the model lack complex dependency logic. Consequently, he does not experience the understanding problems, and the standard pure::variants view showing just the explicitly modeled relations is sufficient.

**Post-interview (RQ2, RQ3).** In the post-interview the four developers, who earlier indicated the need for better support, stated that the relation graph view reduces the effort and perceived difficulty of the model understanding tasks related to implicit feature relations, and helps overcome the model understanding challenges. They indicated that the relation types defined in the view are clearly understandable, and that using the view improves their confidence in the correctness of task results. They stated that the view helps them detect unintended implicit relations and review the intended ones. Importantly, all four developers decided to keep using the graph view in their daily work after the case study.

The fifth developer stated in the post-interview that in his opinion, the relation graph view is useful for complex feature models. However, in the case of his less complex model it does not provide additional insights. The model of this developer contains 184 features and 97 constraints, which create 198 explicit strong relations (and additional 224 tree-based explicit strong relations).

The transitive closure of the graph contains 1324 strong non-tree relations, which are derived from maximally four explicit relations. Hence, the number of transitive edges and the length of derivation chains are much lower than in the example model graph analyzed in Section 5. At present, the model is used in isolation, but connecting it to the system-level model is planned. The developer expressed interest in using the relation graph view again to track inter-model relation chains when working on the connection.

**Conclusion and threats to validity.** The case study provided a positive answer to all three research questions. However, the results indicate that the approach is only beneficial for complex feature models. This is a logical outcome, as for simple models there should be no understanding problems and hence no need for support.

A threat to the external validity (generalizability) of the result is that all participants work in the same company and use the same modeling tool. We slightly mitigated this threat by involving participants from other departments. We addressed the construct validity (adequate representation of cause and effect) by the real-work setting of the case study, as the developers performed their actual work tasks. Internal validity (relation of the treatment to the outcome) was addressed by varying just the one investigated factor, i.e., the use of the relation graph view, between the compared settings.

## 7 Discussion

**Addressing the model understanding challenges.** The transitively closed relation graph contains edges representing all explicit and implicit relations in the feature model. To get an overview over the constraints of a given feature, it is sufficient to review all relation edges of that feature. This information helps address the reported model understanding challenges Q1-Q4, particularly if the feature models are large, complex, or are related to further models.

We believe that the categorization of implication subgraphs into relation types, reducing the complexity of the graph, is the deciding factor that supports model understanding. The low understandability of the original implication graph is indicated by the results of Martinez et al. [13], who found that reasoning about the 8 possible implication relations of two features was confusing for the developers. In contrast to that, the Bosch developers understood the feature relation graph very well. As a second factor, the decomposition of the constraints into the binary form of the individual relations further supports model understanding.

Finally, the relation explanation view, showing how a relation or a problem was derived from the explicit constraints of the model, provides a crucial contribution to model understanding: the derivation information reduces the effort to analyze a particular relation, and increases the user confidence in the analysis result.

**Analysis of the whole model.** While answering questions related to an individual feature is supported well by the relation graph view, questions related to the whole model (e.g., challenge Q5) still require significant effort. For example, the evolution scenario of two related models, described in Section 2, requires that relations of all features are reviewed for changes. In Section 8, we propose an extension of the relation graph view to address this scenario.

**Completeness of the relation graph.** The use of strong and weak relation edges enables the construction of relation graphs for all constraints expressed as Boolean functions over the features.

However, the use of weak edges for human understanding is only practical if the presence conditions are simple. In our experience, the model developers tend to focus on the strong edges when analyzing the model, neglecting the more complex constraints. Furthermore, pure::variants models use the pvSCL constraint language. pvSCL supports Boolean logic, arithmetical and string calculations, collections, tree iterators, variable declarations, and user-defined functions [16]. Consequently, the relation graph cannot fully express the semantics of a pure::variants model. The pvSCL constraints are not parsed by the graph view. Instead, the use of a feature in the given constraint is indicated by a “used in pvSCL” table item, and the constraint interpretation is left to the user. **Performance.** For the example model with 360 features, the transitive graph calculation using a single thread takes 4 seconds on a 2.9 GHz CPU. However, we did not yet optimize the algorithm performance. We plan to investigate fast algorithms for transitive graph closure. For example, Nuutila provides near-linear time algorithms [14]. Note that the graph only needs to be recalculated if the model changes.

## 8 Extension Possibilities

In this section, we list future work ideas of using the relation graph approach in feature modeling and configuration.

**Model evolution support.** Challenge Q5 addresses understanding the impact of changes between two feature model versions (e.g., adding a new constraint). The feature relation graph using strong and weak edges represents all Boolean constraints of the feature model. Hence, two graphs calculated for the old and the new model version can be compared to each other to determine the full extent of the change impact on the implicit model constraints. For example, when editing the model, the developers could be simultaneously presented a list of implicit constraints added, deleted or changed due to the just added explicit constraint. By reviewing the list, they could ensure that no unwanted implicit change happens. The same approach can be used when evolving a group of related feature models. In Section 2, we presented a problem example of two related models, where changes to Model 2 induce implicit constraints in the Model 1. Using the above graph comparison idea, the modeler of Model 2 can immediately spot the unintended impact on Model 1 while adding the new constraints. Alternatively, the modeler of Model 1 can use the graph comparison method when performing an update from the old to the new version of Model 2, and can automatically detect that the new Model 2 version causes new implicit constraints in Model 1.

**Support for feature configuration.** The survey of Rabiser et al. [17] determined that information on feature dependencies is important for the human performing product configuration. To support that goal, the relation graph can be used during configuration to show the users the consequences of the given feature selection before they make the choice. Analogically to the approach of Krieter et al. [11], the current configuration state can be used in each configuration step to reevaluate the presence conditions of weak edges and to remove them or promote them to strong edges.

**Improving the quality of modeled constraints.** Along with finding modeling problems such as dead features, the relation graph can be used to automatically find redundant explicit constraints. For example, the explicit constraint  $A \Rightarrow C \vee D$  does not need to be modeled if constraints  $A \Rightarrow B$  and  $B \Rightarrow C$  exist. In addition, using feature D is

superfluous in this example constraint, as it is anyway satisfied when  $A \Rightarrow C$ . Hence, the graph can help to reduce the number of modeled constraints, and to shorten the existing constraint expressions, without changing the semantics of the model. However, some developers might still tend to retain the redundant constraints in the model for documentation or double-checking purposes.

**Model complexity metrics.** As shown in Section 5, the relation graph can be used to calculate the following feature model metrics:

- number of implicit constraints in the model,
- the length (number of explicit constraints used) and the locality (number of different models used) of their derivations,
- number of implicit constraints for a given feature.

It is interesting to further investigate how these or similar metrics characterize the cognitive complexity of a feature model.

**Reverse engineering the constraints** from existing configurations [13][18] can be supported by abstracting the extracted implication graphs into relation graphs for a better human understanding.

## 9 Conclusion

In this paper, we report on the feature model understanding challenges experienced by model developers at Bosch PS-EC. The challenges are related to the existence of a large number of implicit constraints in the feature models, and to the use of groups of feature models related to each other through inter-model constraints. To counteract the challenges, we define the modal feature relation graphs. The relations abstract the subgraphs of implications between two features, allowing for a better understanding of the resulting logical relation by the developers. We calculate the transitive closure of the relation graph, and use a tabular graph view to support feature model understanding tasks involving implicit constraints. In addition to the constraints, the view also shows modeling problems such as dead, always selected, or false optional features. For every implicit constraint or problem, an explanation tree showing its derivation from the explicit constraints is shown on demand.

In a case study, we received positive feedback from Bosch developers who used the relation graph view. They stated that viewing the implicit relations of a feature helps them solve the model understanding challenges and find modeling problems. We believe that two further factors which support good understandability of the result are the categorization of implication subgraphs into relation types (reducing the complexity of the graph), and the decomposition of the constraints into a binary form (making it easier to reason about the individual relations).

Based on the feedback, we plan to include the relation view as a part of the feature modeling toolkit at Bosch PS-EC. We also plan to implement the view extension ideas: the model evolution support that compares the relation graphs calculated before and after the model change, and the support for model configuration.

## ACKNOWLEDGMENTS

This work is partially supported by the ITEA3 project REVaMP<sup>2</sup>, funded by the BMBF (German Ministry of Research and Education) under grant number 01IS16042C.

We thank the Bosch feature model developers who participated in the case study, and the tool support of pure-systems GmbH for information and clarifications regarding pure::variants APIs.

## REFERENCES

- [1] M. Acher, P. Collet, A. Gaignard, P. Lahire, J. Montagnat, R. France. Composing Multiple Variability Artifacts to Assemble Coherent Workflows. 2012. In Software Quality Journal, Vol. 20, Issue 3-4, September 2012, pp.689-734. DOI: <https://doi.org/10.1007/s11219-011-9170-7>
- [2] D. Benavides, S. Segura, A. Ruiz-Cortes. Automatic Analysis of Feature Models 20 Years Later: A Literature Review. 2010. In Journal Information Systems, Vol. 35, Issue 6, September 2010. DOI: <https://doi.org/10.1145/2791060.2791114>
- [3] T. Berger, R. Rublack, D. Nair, J.M. Atlee, M. Becker, K. Czarnecki, A. Wąsowski. A Survey of Variability Modeling in Industrial Practice. 2013. In Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '13), DOI: <https://doi.org/10.1145/2430502.2430513>
- [4] D. Beuche, R. Hellebrand. Using pure::variants Across the Product Line Lifecycle. 2015. In Proceedings of the 19th International Software Product Line Conference (SPLC 2015), DOI: <https://doi.org/10.1016/j.jis.2010.01.001>
- [5] G. Botterweck, S. Thiel, D. Nestor, S. bin Abid, C. Cawley. Visual Tool Support for Configuring and Understanding Software Product Lines. 2008. In Proceedings of the 12th International Software Product Line Conference (SPLC 2008), DOI: <https://doi.org/10.1109/SPLC.2008.32>
- [6] K. Czarnecki, A. Wąsowski. Feature Diagrams and Logic: There and Back Again. 2007. In Proceedings of the International Software Product Line Conference (SPLC 2007), pp. 23-34, DOI: <https://doi.org/10.1109/SPLINE.2007.24>
- [7] S. El-Sharkawy, S.J. Dhar, A. Krafczyk, S. Duszynski, T. Beichter, K. Schmid. Reverse Engineering Variability in an Industrial Product Line: Observations and Lessons Learned. 2018. In Proceedings of the International Software Product Line Conference (SPLC 2018), DOI: <https://doi.org/10.1145/3233027.3233047>
- [8] A. Jakšić, R. France, P. Collet, S. Ghosh. Evaluating the Usability of a Visual Feature Modeling Notation. 2014. In Proceedings of the 7th International Conference on Software Language Engineering (SLE 2014). DOI: [https://doi.org/10.1007/978-3-319-11245-9\\_7](https://doi.org/10.1007/978-3-319-11245-9_7)
- [9] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. 1990. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University.
- [10] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, I. Schaefer. Is There a Mismatch between Real-World Feature Models and Product Line Research?. 2017. In Proceedings of ESEC/FSE'17, DOI: <https://doi.org/10.1145/3106237.3106252>
- [11] S. Krieter, T. Thüm, S. Schulze, R. Schröter, G. Saake. Propagating Configuration Decisions with Modal Implication Graphs. 2018. In Proceedings of the International Conference on Software Engineering (ICSE 2018), DOI: <https://doi.org/10.1145/3180155.3180159>
- [12] F. van der Linden, K. Schmid, E. Rommes. Philips Medical Systems. In Software Product Lines in Action, Chapter 15, Springer-Verlag Berlin Heidelberg, 2007, DOI: <https://doi.org/10.1007/978-3-540-71437-8>
- [13] J. Martinez, T. Ziadi, R. Mazo, T. F. Bissyande, J. Klein, Y. Le Traon. Feature Relation Graphs: A Visualization Paradigm for Feature Constraints in Software Product Lines. 2014. In Proceedings of the Working Conference on Software Visualization (VISSOFT 2014), DOI: <https://doi.org/10.1109/VISSOFT.2014.18>
- [14] E. Nuutila. Efficient Transitive Closure Computation in Large Digraphs. 1995. PhD Thesis, Mathematics and Computing in Engineering Series No. 74, Finnish Academy of Technology.
- [15] R. Pohl, V. Stricker, K. Pohl. Measuring the Structural Complexity of Feature Models. 2013. In Proceedings of the Intl. Conference on Automated Software Engineering (ASE 2013), DOI: <https://doi.org/10.1109/ASE.2013.6693103>
- [16] pure-systems GmbH. pure::variants User's Guide. Version 4.0.17.685. 2019. Available at: <https://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf> (last accessed on 29 March 2019)
- [17] R. Rabiser, P. Grünbacher, D. Dhungana. Requirements for Product Derivation Support: Results from a Systematic Literature Review and an Expert Survey. 2010. Information and Software Technology, Vol. 52 Issue 3, March 2010, DOI: <https://doi.org/10.1016/j.infsof.2009.11.001>
- [18] S. She, R. Lotufo, T. Berger, A. Wasowski, K. Czarnecki. Reverse Engineering Feature Models. 2011. In Proceedings of the International Conference on Software Engineering (ICSE 2011), pp. 461-470, DOI: <https://doi.org/10.1145/1985793.1985856>
- [19] K. Sierszecki, M. Steffens, H. H. Hojrup, J. Savolainen, D. Beuche. Extending Variability Management to the Next Level. 2014. In Proceedings of the International Software Product Line Conference (SPLC 2014), DOI: <http://dx.doi.org/10.1145/2648511.2648548>
- [20] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stoltz, S. Ferber. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. 2004. In Proceedings of the International Software Product Line Conference (SPLC 2004). DOI: [https://doi.org/10.1007/978-3-540-28630-1\\_3](https://doi.org/10.1007/978-3-540-28630-1_3)
- [21] R. Tartler, D. Lohmann, J. Sincero, W. Schröder-Preikschat. Feature Consistency in Compile-Time-Configurable System Software. 2011. In Proceedings of EuroSys '11. DOI: <https://doi.org/10.1145/1966445.1966451>
- [22] S. Urli, M. Blay-Fornarino, P. Collet. Handling Complex Configurations in Software Product Lines: a Toolied Approach. 2014. In Proceedings of the 18th International Software Product Line Conference (SPLC'14), September 2014. DOI: <https://doi.org/10.1145/2648511.2648523>
- [23] S. Urli, A. Bergel, M. Blay-Fornarino, P. Collet, S. Mosser. A Visual Support for Decomposing Complex Feature Models. 2015. In Proceedings of the 3rd Working Conference on Software Visualization (VISSOFT 2015), September 2015. DOI: <https://doi.org/10.1109/VISSOFT.2015.7332417>

# Second International Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution 2019)

Michael Nieke

Technische Universität Braunschweig  
Braunschweig, Germany  
m.nieke@tu-bs.de

Jacob Krüger

Otto-von-Guericke University  
Magdeburg, Germany  
jkrueger@ovgu.de

Lukas Linsbauer

Johannes Kepler University  
Linz, Austria  
lukas.linsbauer@jku.at

Thomas Leich

Harz University & METOP GmbH  
Wernigerode & Magdeburg, Germany  
leich@hs-harz.de

## ABSTRACT

Most of today's software systems evolve continuously and need to exist in various variants to address different requirements. However, changes resulting from evolution in time (i.e., revisions) and changes resulting from evolution in space (i.e., variants) are managed completely differently. In particular, no traditional technology for version management provides convenient means to effectively and efficiently support unified revision and variant management. Researchers from various communities have proposed new concepts and techniques to tackle this problem. Especially the communities of software configuration management, software product lines, and software versioning are working on such a unified technology. For example, variation control systems have been proposed to systematically manage revisions and variants based on a unified perspective of evolution in time and space. *VariVolution* (the 2nd International Workshop on Variability and Evolution of Software-Intensive Systems) aims to gather researchers and practitioners that are working on or interested in software evolution and variability. The workshop offers an opportunity to exchange ideas, report real-world cases and problems, and initiate new research directions and collaborations.

## CCS CONCEPTS

- **Software and its engineering** → *Software product lines; Software configuration management and version control systems.*

## KEYWORDS

Evolution, variability, version control, configuration management

### ACM Reference Format:

Michael Nieke, Lukas Linsbauer, Jacob Krüger, and Thomas Leich. 2019. Second International Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution 2019). In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342367>

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7138-4/19/09.

<https://doi.org/10.1145/3336294.3342367>

## WORKSHOP SUMMARY

A software product line enables an organization to systematically manage a set of similar software systems based on a reusable platform. However, adopting and maintaining a software product line is a challenging task. An organization needs to actively evolve its software product line to integrate new (extending the variant space) or changing (evolving to a new revision) requirements. Consequently, the two dimensions of evolution in time and space will affect each other, which is why they must be considered mutually. This combination of variability and versioning yields an additional level of complexity. The software-product-line community acknowledges that a unified solution for managing variants and revisions simultaneously is needed to support practitioners. For example, Dagstuhl seminar 19191<sup>1</sup> has been conducted as a collaborative event with researchers from various communities to initiate research on such a solution.

This year's VariVolution workshop follows the topics of the previous edition [1] and conveys concepts of software evolution to the variability management community. The objectives of the workshop are:

- Discuss recent research that addresses the challenges of variability and evolution.
- Identify open research topics.
- Present concepts and technical solutions towards unified variant and revision management.
- Demonstrate tools addressing the aforementioned topics.
- Investigate real-world problems caused by the combination of variability and evolution.
- Present industrial challenges and lessons learned.

VariVolution aims to increase collaboration in the domain of variability evolution and to raise awareness of current problems in this domain. The second edition of VariVolution received six submissions, out of which the program committee accepted four full papers and one short paper to be included in the workshop.

**Workshop website:** [sites.google.com/view/varivolution2019](http://sites.google.com/view/varivolution2019)

## REFERENCES

- [1] Lukas Linsbauer, Somayeh Malakuti, Andrey Sadovsky, and Felix Schwägerl. 2018. 1st Intl. Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution). In *International Systems and Software Product Line Conference (SPLC)*. ACM, 294–294.

---

<sup>1</sup><https://www.dagstuhl.de/en/program/calendar/semp?semnr=19191>

# Second International Workshop on Experiences and Empirical Studies on Software Reuse (WEESR 2019)

Jaime Chavarriaga  
Universidad de los Andes  
Bogotá, Colombia  
ja.chavarriaga908@uniandes.edu.co

Julio Ariel Hurtado  
Universidad del Cauca  
Popayán, Colombia  
ahurtado@unicauca.edu.co

## ABSTRACT

The *Workshop on Experiences and Empirical Studies on Software Reuse (WEESR)* aims, on the one hand, to allow researchers and practitioners discuss in-progress research regarding experiences and empirical studies on applying reuse techniques in non-academic environments. On the other hand, it aims for providing feedback on how these studies are planned, designed, conducted, and reported. The second edition of this workshop, the WEESR 2019, was co-located with the Software Product Lines Conference at 2019 (SPLC'19). There, attendants discussed six papers presenting case studies, GQM experiments, empirical studies and evaluations regarding proposals on topics such as reuse in companies not aware of software product lines, companies adopting product lines, defining the scope of a product line, modeling variability in self-adaptive systems, and reusing software for implementing artificial neural networks.

## CCS CONCEPTS

- Software and its engineering → Software product lines.

## KEYWORDS

Software reuse, Empirical Software Engineering

### ACM Reference Format:

Jaime Chavarriaga and Julio Ariel Hurtado. 2019. Second International Workshop on Experiences and Empirical Studies on Software Reuse (WEESR 2019). In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342366>

## 1 MOTIVATION

Software Reuse has been recognized as a key strategy to improve productivity and quality. For instance, during many years, technologies such as components, object orientation, and product lines have been proposed to promote reuse and achieve these benefits. However, although these technologies may be enablers, recent studies show that non-technical factors, such as organization, processes, and human involvement, appeared to be at least as important [1][2][3]. It is becoming very important understand how these

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7138-4/19/09.

<https://doi.org/10.1145/3336294.3342366>

non-technical factors may affect in-practice existing proposals to achieve systematic reuse in software organizations.

The *Second Workshop on Experiences and Empirical Studies on Software Reuse (WEESR 2019)*, co-located in the Software Product Line Conference (SPLC'19), aimed at discussing experiences and studies in the area of software reuse, the challenges to overcome to perform these studies and possible improvements for these works.

## 2 SUMMARY OF CONTRIBUTIONS

*Variability Management on companies not aware of software product lines.* Chacon-Luna et al. presented an evaluation of reuse practices in one Ecuadorian company. The authors found practices for reusing source code but not for modeling variability or considering variability in analysis. They plan to extend their study by evaluating more companies.

*Analyzing the convenience of adopting software product lines.* Rincón et al. discussed an evaluation of their APPLIES framework to analyze benefits, level of preparation and possible issues for companies considering to implement software product lines. The results of this evaluation has been used to improve the framework.

*Collaborative practices at scoping product lines.* Camacho et al. presented their study on the collaborative tasks required at scoping a product line. They use Thinklets, i.e., design patterns for collaborative tasks, to propose improvements.

*Modeling variability in Self-adaptive systems.* Achtaich et al. discusses their evaluation of the State-Constraint Transition (SCT) modelling language, a technique used to model requirements for Self-adaptive systems.

*Modeling Variability in Accessibility requirements.* Rodriguez et al. presented a set of variability models representing the Web Content Accessibility Guidelines (WCAG 2.1) and its application to evaluate MOOC sites.

*Reusability in Artificial Neural Networks.* Ghofrani et al. interviewed experts on Software Engineering and on Artificial Neural Networks (ANNs) to analyze reuse practices. They found a low level of reuse. In addition, much of the participants, 79 out of 112, considered reuse as the modification of an existing ANN to adapt it to a new context, instead to designing and using reusable modules or libraries.

## REFERENCES

- [1] Cagatay Catal. 2009. Barriers to the adoption of software product line engineering. *ACM SIGSOFT Software Engineering Notes* 34, 6 (2009), 1.
- [2] J Ferreira Bastos, P. Anselmo da Mota Silveira Neto, Eduardo Santana de Almeida, and S. Romero de Lemos Meira. 2011. Adopting software product lines: a systematic mapping study. In *15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011)*. IET, Durham, UK, 11–20.
- [3] Karma Sherif and Ajay Vinze. 2003. Barriers to Adoption of Software Reuse a Qualitative Study. *Information and Management* 41, 2 (Dec. 2003), 159–175.

# Fourth International Workshop on Software Product Line Teaching (SPLTea 2019)

Mathieu Acher

Univ Rennes, Inria, CNRS, IRISA  
Rennes, France  
mathieu.acher@irisa.fr

Rick Rabiser

Johannes Kepler University Linz  
Linz, Austria  
rick.rabiser@jku.at

Roberto E. Lopez-Herrejon

Ecole de Technologie Supérieure  
Montréal, Canada  
Roberto.Lopez@etsmtl.ca

## ABSTRACT

Education has a key role to play for disseminating the constantly growing body of Software Product Line (SPL) knowledge. In a sense, every researcher in SPL should think about how to teach SPL. This workshop aims to explore and explain the current status and ongoing work on teaching SPLs at universities, colleges, and in industry (e.g., by consultants). This fourth edition will continue the effort made at SPLTea'14, SPLTea'15 and SPLTea'18. In particular we seek to better understand how to build a curriculum for teaching SPLs – a central issue as reported in surveys and as informally discussed at SPLTea'18. We expect several lightning talks that report on traditional questions like: what is the targeted audience? What is the place in the curriculum? What is the material (slides, tools, books, etc) used? As there is hardly a one-size-fits-all curriculum, the workshop aims to collectively identify commonality and variability when building SPL curriculums. As a concrete outcome, we expect to elaborate a variability model of SPL teaching that could be actuated to derive custom curriculum in various contexts.

## CCS CONCEPTS

• Software and its engineering → Software product lines.

## KEYWORDS

Software product lines, teaching, education

### ACM Reference Format:

Mathieu Acher, Rick Rabiser, and Roberto E. Lopez-Herrejon. 2019. Fourth International Workshop on Software Product Line Teaching (SPLTea 2019). In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342363>

## SPLTEA WORKSHOP

With over two decades of existence, Software Product Lines (SPLs) are now well-established in research and industry [2, 3]. The body of knowledge collected and organized by the SPL research community is still growing. However, without any effort for disseminating this knowledge, engineers of tomorrow are unlikely to be aware of the issues faced when engineering SPLs (or configurable systems) – up to the point they will not recognize such systems. In turn, they will

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7138-4/19/09.

<https://doi.org/10.1145/3336294.3342363>

not use appropriate techniques and face problems such as scalability that the SPL community perhaps already studied or solved.

We believe education has a key role to play. The teaching of SPLs can enable the next generation of engineers to build highly complex, adaptive, and configurable software systems. Also, research can benefit from teaching: students can be involved in controlled experiments and researchers involved in teaching can identify potential missing gaps of SPL engineering tools and techniques. Teaching SPLs is challenging. Currently, it is unclear how SPLs are taught, what are the possible gaps and difficulties faced, the benefits, or the material available. To address this gap, we conducted several surveys [1] to capture a snapshot of the state of teaching in our community. Our goal was to identify common threads, interests, and problems and build upon them to further understand and hopefully strengthen this important need in our community.

The results as well as the participation and discussions at VaMoS'2014, SPLTea'14, SPLTea'15, and SPLTea'18 motivated us to propose another teaching workshop. SPLTea'14 attracted 30+ participants. SPLTea'15 focused on the design and population of an open repository of resources dedicated to SPL teaching.

<http://teaching.variability.io>

The third edition (SPLTea'18) attracted 20+ participants with 3 talks (experience reports) and several discussions, for example: how to teach SPLs to undergraduates, "briefly"? How to teach SPL core concepts (and what are they, actually) to non-software experts?

This fourth edition, SPLTea 2019, will put the focus on *how to build a curriculum* for teaching SPLs because this was an important issue discussed at SPLTea'18 requiring further discussions. We expect several (lightning) talks that report on traditional questions like: what is the targeted audience for SPL teaching? What is the place for SPLs in the curriculum? What is the material (slides, tools, books, etc) used? What are the benefits of teaching SPLs? What are the difficulties and barriers? Another (the main) objective of the workshop is to collectively *elaborate a variability model of SPL teaching*. We will have a specific session for identifying (1) commonalities and core concepts that can be reused in any SPL curriculum (2) variability and particularities since customization is needed to address the various requirements and constraints of (SPL) teaching contexts and existing curricula.

## REFERENCES

- [1] Mathieu Acher, Roberto E. Lopez-Herrejon, and Rick Rabiser. 2017. Teaching Software Product Lines: A Snapshot of Current Practices and Challenges. *ACM TOCE* 18, 1 (2017), 2:1–2:31.
- [2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.
- [3] Frank van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering*. Springer.

# First International Workshop on Languages for Modelling Variability (MODEVAR 2019)

David Benavides  
University of Sevilla  
Sevilla, Spain  
benavides@us.es

Don Batory  
University of Texas at Austin  
Austin, USA  
batory@cs.utexas.edu

Rick Rabiser  
Johannes Kepler University Linz  
Linz, Austria  
rick.rabiser@jku.at

Mathieu Acher  
Univ Rennes, Inria, CNRS, IRISA  
Rennes, France  
mathieu.acher@irisa.fr

## ABSTRACT

Feature models were invented in 1990 and have been recognised as one of the main contributions to the software product line community. Although there have been several attempts to establish a sort of standard variability modelling language, there is still no consensus. There can be many motivations to have one but there is one that is very important: information sharing among researchers, tools or developers. This first international workshop is an interactive event where all participants shall share knowledge about how to build up a simple variability modelling language that all the community can agree on.

## CCS CONCEPTS

- Software and its engineering → Software product lines.

## KEYWORDS

Software product lines, variability modelling, languages

### ACM Reference Format:

David Benavides, Rick Rabiser, Don Batory, and Mathieu Acher. 2019. First International Workshop on Languages for Modelling Variability (MODEVAR 2019). In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342364>

## MODEVAR WORKSHOP

The main topic of discussion of this workshop is variability modelling languages, concepts, usages and tools. In 1990 [5], feature models were proposed as a way of modelling variability. Since then, this contribution is among the most important ones in the software product line history. Nevertheless, there have been many feature modelling dialects that were well surveyed by some authors. Czarnecki et al. [2] provide a good historical perspective. Also, diverse other approaches not focusing on features as unit of variability exist, e.g., decision modelling [8], OVM [6], and UML-based

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7138-4/19/09.

<https://doi.org/10.1145/3336294.3342364>

variability modelling [3], to name the probably most prominent. Diverse (systematic) studies have surveyed existing variability modelling languages and approaches. Raatikainen et al. [7] published a tertiary study demonstrating the plethora of existing approaches. Bashroush et al. [1] present a survey of 37 tools for variability management/modelling.

However, no standard emerged. We cannot share easily our models and we cannot have a sort of universal variability model repository where we could post our outputs using a common notation. Other communities do have such kind of languages such as DIMACS for SAT solving or XCSP for constraint programming to only mention a few. CVL [4] was proposed as a try to have a standard and common variability language with many different purposes. The project unfortunately did not end up very well. Recently, a family of industrial standards were released such as ISO/IEC 26558:2017. As a community, we are missing a simple common variability modelling language for academic purposes as a first output. From there, we expect to advance the state of the art not only in modelling variability but also in today's variability managers.

In this context, this workshop intends to join forces to make what could be an important contribution to the community. It aims to reach a consensus on the main modelling constructs and the language's syntax and have a big enough and representative consortium that supports the proposal.

## REFERENCES

- [1] Rabih Bashroush, Muhammad Garba, Rick Rabiser, Iris Groher, and Goetz Botterweck. 2017. Case tool support for variability management in software product lines. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 14.
- [2] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool features and tough decisions: a comparison of variability modeling approaches. In *6th Int'l WS on Variability Modelling of Software-Intensive Systems*. ACM, 173–182.
- [3] Hassan Gomaa. 2005. *Designing Software Product Lines with UML*. Addison-Wesley.
- [4] Øystein Haugen, Andrzej Wąsowski, and Krzysztof Czarnecki. 2013. CVL: common variability language. In *17th Int'l Software Product Line Conf.* ACM, 277–277.
- [5] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. SEI, Carnegie-Mellon Univ.
- [6] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer.
- [7] Mikko Raatikainen, Juha Tiilonen, and Tomi Männistö. 2019. Software product lines and variability modeling: A tertiary study. *Journal of Systems and Software* 149 (2019), 485–510.
- [8] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. 2011. A comparison of decision modeling approaches in product lines. In *5th Int'l WS on Variability Modelling of Software-Intensive Systems*. ACM, 119–126.

# Seventh International Workshop on Reverse Variability Engineering (REVE 2019)

Mathieu Acher

Univ Rennes, Inria, CNRS, IRISA  
Rennes, France  
mathieu.acher@irisa.fr

Roberto E. Lopez-Herrejon  
Ecole de technologie supérieure  
Montréal, Canada  
roberto.lopez@etsmtl.ca

## ABSTRACT

Software Product Line (SPL) migration remains a challenging endeavour. From organizational issues to purely technical challenges, there is a wide range of barriers that complicates SPL adoption. This workshop aims to foster research about making the most of the two main inputs for SPL migration: 1) domain knowledge and 2) legacy assets. Domain knowledge, usually implicit and spread across an organization, is key to define the SPL scope and to validate the variability model and its semantics. At the technical level, domain expertise is also needed to create or extract the reusable software components. Legacy assets can be, for instance, similar product variants (e.g., requirements, models, source code etc.) that were implemented using ad-hoc reuse techniques such as clone-and-own. More generally, the workshop REverse Variability Engineering (REVE) attracts researchers and practitioners contributing to processes, techniques, tools, or empirical studies related to the automatic, semi-automatic or manual extraction or refinement of SPL assets.

## CCS CONCEPTS

- Computer systems organization → Embedded systems; Redundancy; Robotics;
- Networks → Network reliability.

## KEYWORDS

datasets, neural networks, gaze detection, text tagging

### ACM Reference Format:

Mathieu Acher, Tewfik Ziadi, Roberto E. Lopez-Herrejon, and Jabier Martinez. 2019. Seventh International Workshop on Reverse Variability Engineering (REVE 2019). In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342365>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3342365>

Tewfik Ziadi

Sorbonne Université, CNRS, LIP6  
Paris, France  
[tewfik.ziadi@lip6.fr](mailto:tewfik.ziadi@lip6.fr)

Jabier Martinez

Tecnalia  
Derio, Spain  
[jabier.martinez@tecnalia.com](mailto:jabier.martinez@tecnalia.com)

## REVE WORKSHOP

The adoption of a Software Product Line (SPL) architecture or process is not without obstacles. A common approach is to leverage existing legacy assets and support an extractive adoption [1]. Those existing assets are typically similar variants that were implemented using ad-hoc reuse techniques such as copy-paste-modify. Variability can also be hidden and scattered in numerous strongly coupled artefacts. Mining the variability of these assets could reduce the costs and risks of the migration to SPL engineering practices. However, there are several challenges to address, depending on the kinds of artefacts and the engineering contexts. Several approaches to automatically or semi-automatically extract variability continue to be proposed, applied, validated and improved. This might include but not limited to feature identification and location, feature constraints discovery, feature model synthesis or reusable assets extraction. The Reverse Variability Engineering workshop, following the topics of the previous editions (e.g., [2, 3]), brings the reverse engineering and reengineering communities to variability management issues. Website: <http://reveworkshop.github.io>. The objectives of the workshop are to:

- provide a meeting point for researchers and practitioners in the area;
- review and formulate a research agenda in reverse engineering for variability;
- identify and gather a corpus of case studies and benchmarks to benefit the research and practitioner community.

REVE aims to increase collaborative works on this research topic and to improve the mining techniques for the benefit of the SPL and related communities.

## REFERENCES

- [1] CharlesW Krueger. 2001. Easing the transition to software mass customization. In *International Workshop on Software Product-Family Engineering*. Springer, 282–293.
- [2] Roberto Erick Lopez-Herrejon, Jabier Martinez, Tewfik Ziadi, and Mathieu Acher. 2016. Fourth international workshop on reverse variability engineering (REVE 2016). In *Proceedings of the 20th International Systems and Software Product Line Conference, SPLC 2016, Beijing, China, September 16–23, 2016*. 345. <https://doi.org/10.1145/2934466.2962734>
- [3] Jabier Martinez, Roberto E. Lopez-Herrejon, Tewfik Ziadi, and Mathieu Acher. 2017. REVE 2017: 5th International Workshop on REverse Variability Engineering. In *Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume A, Sevilla, Spain, September 25–29, 2017*. 245. <https://doi.org/10.1145/3106195.3106199>

# Machine Learning and Configurable Systems: A Gentle Introduction

Hugo Martin, Juliana Alves Pereira,  
Mathieu Acher  
Univ Rennes, IRISA, Inria, CNRS, France

Paul Temple  
PReCISE, NaDi  
University of Namur, Belgium

## ABSTRACT

The goal of this tutorial is to give an introduction to how machine learning can be used to support activities related to the engineering of configurable systems and software product lines. To the best of our knowledge, this is the first practical tutorial in this trending field. The tutorial is based on a systematic literature review and includes practical tasks (specialization, performance prediction) on real-world systems (VaryLaTeX, x264).

## CCS CONCEPTS

- Software and its engineering → Software product lines;

## KEYWORDS

Software Product Lines, Machine Learning, Configurable Systems

### ACM Reference format:

Hugo Martin, Juliana Alves Pereira,

Mathieu Acher and Paul Temple. 2019. Machine Learning and Configurable Systems: A Gentle Introduction. In *Proceedings of 23rd International Systems and Software Product Line Conference - Volume A, Paris, France, September 9–13, 2019 (SPLC '19)*, 2 pages.

DOI: 10.1145/3336294.3342383

Configurable software systems and software product lines allow stakeholders to derive product variants that meet their specific functional and non-functional requirements. A straightforward way to find a suitable configuration is to measure the target (non-functional) property of each corresponding variant and then select any configuration that meets specified requirements. Unfortunately, enumerating and measuring all configurations is usually unfeasible due to the combinatorial explosion of possible variants and the cost of measurements. Machine learning techniques have gained momentum to predict the properties of configurable systems out of a sample – without measuring all variants. In this tutorial, we rely on the pattern "*sampling, measuring, learning, validation*" emerged in the software engineering and machine learning literature [3]. The usual process is to sample some configurations, execute and measure e.g., their execution time, and learn out of measurements.

The practical tasks will be conducted on two case studies: VaryLaTeX [1] and x264 [2]. *VaryLaTeX* is a learning system capable of generating LaTeX paper variants that respect constraints (e.g.,

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC '19, Paris, France

© 2019 ACM 978-1-4503-7138-4/19/09...\$15.00

DOI: 10.1145/3336294.3342383

page limits). The VaryLaTeX case is used to illustrate how learning techniques can be used to *specialize* configurable systems through the automatic mining of constraints among options. We also use the video encoder *x264* to show how we can predict performance properties of unmeasured configurations [2, 3].

The interactive tutorial is a half-day event structured as follows:

**Part 1 (theoretical and interactive) – Motivation and a complete overview of the progress made in this field.** Starting with the Linux kernel, attendees will understand how impossible it is to enumerate all possible product variants and the need to use alternative approaches, namely statistical machine learning. Based on our recent systematic literature survey [3], we will introduce a catalog of "*sampling, measurement, learning, validation*" techniques spanning different use-cases: performance prediction, configuration optimization, software understanding or specialization (constraint mining). Finally, we give concrete examples to illustrate the underlying challenges (e.g., resources cost vs error cost).

**Part 2 (practical) – Learning constraints: the case of VaryLaTeX.** In this practical session, we exercise how configurable systems can be *specialized* thanks to learning techniques used to automatically mine constraints among options. We use an intuitive example: a learning system, called VaryLaTeX [1]. VaryLaTeX is a solution based on variability, constraint programming, and machine learning techniques for documents written in LaTeX to meet constraints and deliver on time. This part aims to illustrate the end-to-end process (sampling, measuring, learning, validation) but also how to read and interpret decision trees which are used to specialize configurable systems.

**Part 3 (practical) – Performance prediction: the case of x264.** Performance prediction is a real problem on its own because of the complexity of configuration space and approximation made by machine learning techniques. To illustrate the problem, we use x264 a highly configurable video encoder considered in numerous works (see e.g., [2, 3]). Specifically, we show how we can predict performance properties (e.g., execution time) of unmeasured configurations of x264. With this part, attendees will understand how to frame a performance prediction problem as a regression problem and how machine learning algorithms can be applied.

**Part 4 (theoretical) – Conclusion.** We will recap all lessons learned, and point out limitations and open challenges that need attention in future work (e.g., how to select a significant sample of configurations? what is an ideal sample?).

The material (including slides, data, procedures) is available online: <https://github.com/VaryVary/ML-configurable-SPLCTutorial/>

**Acknowledgements.** This research was partially funded by the ANR-17-CE25-0010-01 VaryVary project.

## REFERENCES

- [1] Mathieu Acher, Paul Temple, Jean-Marc Jézéquel, José A. Galindo, Jabier Martínez, and Tewfik Ziadi. 2018. VaryLATEX: Learning Paper Variants That Meet Constraints. In *VAMOS*. 83–88. DOI : <http://dx.doi.org/10.1145/3168365.3168372>
- [2] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *ASE*.
- [3] Juliana Alves Pereira, Hugo Martin, Mathieu Acher, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2019. Learning Software Configuration Spaces: A Systematic Literature Review. (2019). arXiv:arXiv:1906.03018

# Software Reuse for Mass Customization

Mike Mannion

Department of Computing

Glasgow Caledonian University

Glasgow, UK

[m.a.g.mannion@gcu.ac.uk](mailto:m.a.g.mannion@gcu.ac.uk)

Hermann Kaindl

Institute of Computer Technology

TU Wien

Vienna, Austria

[kaindl@ict.tuwien.ac.at](mailto:kaindl@ict.tuwien.ac.at)

## ABSTRACT

This tutorial explores the impact of the socio-economic trends of mass customization on software reuse through software product line development.

## CCS CONCEPTS

- Software and its Engineering → Software Creation and Management → Software Development Techniques → Reusability → Software Product Lines.

## KEYWORDS

Software Reuse, Mass Customization, Product Lines

### ACM Reference format:

Mike Mannion, Hermann Kaindl. 2019. Software Reuse for Mass Customization. In *Proceedings of 23rd International Systems and Software Product Line Conference (SPLC'19), September 9-13 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342378>

## 1 Introduction

Several socio-economic trends are driving customer demands towards individualization. Many suppliers are responding by offering supplier-led software design choices (“mass customization”). Some are also offering customer-led software product design choices (“mass personalization”). We will introduce these concepts and explore the implications on software reuse through software product line development. We will discuss two different approaches. One is grounded in feature modelling, the other in case-based reasoning. Both support the identification and selection of similar products. However, they place different emphases on these activities, use different product descriptions and deploy different product derivation methods. Accordingly, they each have different properties, benefits and limitations.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SPLC'19, September 9-13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s).

ACM-ISBN 978-1-4503-7138-4/19/09

<https://doi.org/10.1145/3336294.3342378>

## 2 Trends in Software Product Development

While object-oriented approaches [1, 2] have become stable, software product line engineering (SPLE) [3, 4] continues to evolve. As customer product demands move toward individualization, many organizations’ development process models have switched to agile and incremental delivery. These process models often make use of software product line and variability management techniques, yet many outstanding challenges remain in matching product supply and customer demand.

## 3 Feature Model Development

A common approach to SPLE is to construct feature models that include the feature variability of the products to be supplied. New but similar products are derived through configuring the feature model by making selections at points of variability.

## 4 Case-based Reasoning

An alternative to feature modelling is case-based reasoning, which is based on retrieving the most similar previous case to the problem to be solved e.g. [5]. New product development is grounded in adapting this case to build a solution.

## 5 Similarity

Evaluating similarity in SPLE is gaining attention e.g. [5, 6, 7]. Different approaches focus on different aspects of the lifecycle and use different similarity models and metrics.

## REFERENCES

- [1] H. Kaindl, Object-Oriented Approaches in Software Engineering and Artificial Intelligence, *Journal of Object-Oriented Programming (JOOP)*, vol. 6, no. 8, 1994, 38–45.
- [2] H. Kaindl, Is object-oriented requirements engineering of interest?. *Requirements Engineering*, vol. 10, 2005, 81–84.
- [3] M. Mannion, O. Lewis, H. Kaindl, G. Montroni, J. Wheadon, Representing Requirements of Generic Software in an Application Family Model, in *Proceedings of 6<sup>th</sup> International Conference on Software Reuse (ICSR-6)*, Vienna, Austria, 27-29 June 2000, LNCS 1844, 153–159.
- [4] M. Mannion, H. Kaindl, Using Parameters and Discriminants for Product Line Requirements. *Systems Engineering*, vol. 11, no. 1, 2008, 61–80.
- [5] H. Kaindl, M. Smialek, W. Nowakowski, Case-based Reuse with Partial Requirements Specifications, in *Proceedings of the 18th IEEE International Requirements Engineering Conference (RE'10)*, 2010, 399–400.
- [6] H. Kaindl, M. Mannion, A Feature-Similarity Model for Product Line Engineering, in *Proceedings of the 14th International Conference on Software Reuse (ICSR 2015)*, *Software Reuse for Dynamic Systems in the Cloud and Beyond*, LNCS 8919, 2014, 34–41.
- [7] M. Al-Hajjiji, M. Schulze, U. Ryssel, Similarity of Product Line Variants, in *Proceedings of the 22<sup>nd</sup> International Systems and Software Product Line Conference*, September 10–14, Gothenburg, Sweden, 2018, 226–235.

# Variability Modeling and Implementation with EASy-Producer

Klaus Schmid, Holger Eichelberger, and Sascha El-Sharkawy

University of Hildesheim, Institute of Computer Science

Hildesheim, Germany

{schmid,eichelberger,elscha}@sse.uni-hildesheim.de

## ABSTRACT

EASy-Producer is an open-source research toolset for engineering product lines, variability-rich software ecosystems, and dynamic software product lines. In this tutorial, we will introduce its (textual) variability modeling capabilities realized by the Integrated Variability Modeling Language (IVML) and its model-based development and implementation capabilities, which are realized by the Variability Instantiation Language (VIL) and the Variability Template Language (VTL).

## CCS CONCEPTS

- Software and its engineering → Software product lines; Model-driven software engineering.

## KEYWORDS

Software product lines, variability modeling, model-based engineering

### ACM Reference Format:

Klaus Schmid, Holger Eichelberger, and Sascha El-Sharkawy. 2019. Variability Modeling and Implementation with EASy-Producer. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342382>

## 1 OVERVIEW

The tutorial introduces the variability modeling capabilities of the Integrated Variability Modeling Language (IVML) [3], which is part of the EASy-Producer product line environment. EASy-Producer is an open-source research toolset for engineering product lines, variability-rich software ecosystems and dynamic software product lines. It has been applied in several industrial case studies and research projects showing its practical applicability both from a stability and a capability point of view. The toolset consists of an interactive approach to product line definition and configuration through DSLs. The tutorial will introduce its (textual) variability modeling capabilities realized by the Integrated Variability Modeling Language (IVML) and its model-based development and implementation capabilities, which are realized by the Variability Instantiation Language (VIL) and the Variability Template Language (VTL). As an outcome, the participants of the tutorial will understand the capabilities and design decisions of the toolset and gain a basic practical

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7138-4/19/09.

<https://doi.org/10.1145/3336294.3342382>

understanding of how to use it to define variability models, constraints, product configurations, and how to implement variability.

While most work on variability modeling centers around interactive modeling, the use of DSLs as a vehicle for variability modeling has significant power to unify the area of product line engineering with model-based development. The fundamental approach of IVML is to allow users to model variability in a very simple fashion (comparable to feature models), while providing extensive powers that encompass the whole range of modeling capabilities as present in model-based development. This allows to handle classical product line engineering as well as model-based development in an integrated fashion. Moreover, capabilities of IVML enable the handling of configuration modeling across multiple products, enabling multi-product line and ecosystem scenarios.

Besides those capabilities, there are further features [4]: First, there is no strong separation between the product line infrastructure and the derived product as opposed to what is common in the product line area. Rather, EASy-Producer centers around the concept of a product line project, which is at the heart of the infrastructure. Second, the tool supports external tools and diverse artifact types that can be modified or generated without modifications on its grammar. As a consequence, the tool aims to be very open to the integration of further tools and to instantiate arbitrary artifacts.

EASy-Producer has been developed over the last couple of years to support complex variability-intensive product lines. Recently, we applied EASy-Producer to topological variability [2], model-based product line instantiation [1], continuous integration, and runtime variations. While being open source on GITHUB, the tool can be used on industrial product lines as we have shown in some case studies, making it particularly interesting to practitioners. Nightly-builds as well as the specifications of the languages are available on our update site: <https://projects.sse.uni-hildesheim.de/easy/>

## ACKNOWLEDGMENTS

We thank Christian Kröher, Roman Sizonenko, Cui Qin, Aike Sass, and Bartu Dernek for their contributions.

This work is partially supported by the ITEA3 project REVaMP<sup>2</sup>, funded by the BMBF (German Ministry of Research and Education) under grant 01IS16042H. Any opinions expressed herein are solely by the authors and not by the BMBF.

## REFERENCES

- [1] H. Eichelberger, C. Qin, and K. Schmid. 2017. Experiences with the Model-based Generation of Big Data Pipelines. In *Datenbanksysteme für Business, Technologie und Web (BTW '17) – Workshops*, 49–56.
- [2] H. Eichelberger, C. Qin, R. Sizonenko, and K. Schmid. 2016. Using IVML to Model the Topology of Big Data Processing Pipelines. In *International Systems and Software Product Line Conference (SPLC '16)*, 204–208.
- [3] H. Eichelberger and K. Schmid. 2015. Mapping the Design-space of Textual Variability Modeling Languages: A Refined Analysis. *International Journal on Software Tools for Technology Transfer* 17, 5 (Oct. 2015), 559–584.
- [4] K. Schmid. 2017. EASy-Producer – An Open Toolset for Lightweight Product Line Engineering.

# Describing Variability with Domain-Specific Languages and Models

Juha-Pekka Tolvanen

MetaCase

Jyväskylä, Finland

jpt@metacase.com

Steven Kelly

MetaCase

Jyväskylä, Finland

stevek@metacase.com

## ABSTRACT

This tutorial will teach participants about domain-specific languages and models, where they can best be used (and where not), and how to apply them effectively to improve the speed and quality of product development within a product line.

## CCS CONCEPTS

- Software and its engineering → Software product lines; Domain specific languages; Software configuration management and version control systems; Model-driven software engineering.

## KEYWORDS

Tutorial, domain-specific language, domain-specific modeling, product line variability, product derivation

### ACM Reference Format:

Juha-Pekka Tolvanen and Steven Kelly. 2019. Describing Variability with Domain-Specific Languages and Models. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342377>

## 1 INTRODUCTION

Variability is more than alternative or optional items. It also deals with behavior, ordering and various dependencies – often across several domains such as hardware and software. Domain-Specific Languages (DSLs) can cover this rich variation space, matching and exceeding the capabilities of feature modeling and parameter tables [1]. With a DSL, the language definition itself defines the variation space and related rules, and language users follow this variation space when using the language to create variants. Domain engineers define the language and application engineers use it. In a fair number of cases the final products can be automatically generated from the high-level variant specifications.

Moving to language-based variation allows the development of variants based on common assets that would not be possible with other approaches. It also makes possible the creation of new variability that is not possible with other variability handling mechanisms [1, 3]. Perhaps even more importantly it allows application

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SPLC '19, September 9–13, 2019, Paris, France

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7138-4/19/09...\$15.00

<https://doi.org/10.1145/3336294.3342377>

engineers to directly use the product's own concepts in the specification models and apply their own existing terminology for the domain.

## 2 TUTORIAL CONTENT

This tutorial introduces Domain-Specific Languages and models and looks at how they extend the other variability modeling approaches [4]. This is followed by real-life examples (e.g. [2, 5]) from various industries and product lines, such as industry automation, smart watches, IoT devices and automotive systems. The main part of the tutorial addresses the guidelines for defining DSLs for product lines and for defining the generators that can automatically produce product variants.

On the language creation side, success depends largely on identifying the variation space. Preparing for language definition comes down to conducting a thorough domain analysis. Domain engineers identify the abstractions for variation, the parts which are the same for all products within the product line, and the parts which can vary. Typically the key focus of language engineers is then to define a language which makes it easy to describe the varying parts within a given product line. In the tutorial we inspect some typical DSL patterns that are found from industry cases of product lines. On the variant generation side we look at different ways to implement generators to read the models and produce the full variant code or configuration. Some of these will be demonstrated with running examples covering language and generator development as well as their use.

The target audience for the tutorial includes software and system architects, researchers, product managers, technology managers, consultants, and senior designers. The required expertise level is intermediate and no previous experience with modeling or code generation is needed.

## REFERENCES

- [1] Czarnecki, K., Eisenecker, U., Generative Programming, Methods, Tools, and Applications, Addison-Wesley, 2000.
- [2] Kelly, S., Tolvanen, J.-P., Domain-Specific Modeling: Enabling Full Code Generation, Wiley-IEEE Computer Society Press, 2008
- [3] Tolvanen, J.-P., Kelly, S., Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences. Proceedings of the 9th International Software Product Line Conference, H. Obbink, K. Pohl, Springer-Verlag, LNCS 3714, 2005.
- [4] Tolvanen, J.-P., Kelly, S., How Domain-Specific Modeling Languages Address Variability in Product Line Development: Investigation of 23 cases. Proceedings of 23rd International Systems and Software Product Line Conference (SPLC19). ACM, 2019.
- [5] Tolvanen, J.-P. and Kelly, S. Model-Driven Development Challenges and Solutions – Experiences with Domain-Specific Modelling in Industry. In Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development, SCITEPRESS Science and Technology Publications, Lda, 2016

# Automated Evaluation of Embedded-System Design Alternatives

Maxime Cordy

SnT, University of Luxembourg

maxime.cordy@uni.lu

Sami Lazreg

Université Côte d'Azur, CNRS, I3S, France

lazreg@i3s.unice.fr

## ABSTRACT

This half-day tutorial presents a method to tackle the issue of evaluating a plethora of embedded system design-alternatives against functional and non-functional requirements. Our method results from a joint research project between three universities and Visteon Electronics, a multinational company active in the engineering of automotive embedded systems. We will illustrate its application on a real-world application and, first and foremost, under a practical prism. As such, practitioners developing embedded systems are invited to discover the capabilities of our techniques and tools, as well as insights on how to integrate them into their engineering processes. Moreover, researchers will get a comprehensive picture of the underlying techniques and the challenges that remain ahead.

## CCS CONCEPTS

- Computer systems organization → Embedded software; • Hardware → Model checking; • Software and its engineering → Software product lines.

## KEYWORDS

Embedded systems, Variability-intensive systems, Quality requirements, Model-based evaluation, Design space exploration

### ACM Reference Format:

Maxime Cordy and Sami Lazreg. 2019. Automated Evaluation of Embedded-System Design Alternatives. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342379>

## MOTIVATION AND CONTEXT

The scale and complexity of software-intensive systems such as cyber-physical systems and large-scale embedded systems have reached historic levels. While the Gartner Group expects more than twenty billion of connected objects, automotive systems are developed with several millions of lines of code driving hundreds of electronic hardware. In many of these systems, requirements engineering and design activities are of fundamental importance in the industry to reduce a wide range of risks at an early stage of development. These development steps are, moreover, tightly intertwined and involve complex multi-criteria decision-making over a variety of concerns.

As an example, let us consider an infotainment feature in an automotive system. Specifying such a feature typically entails defining a set of functional and non-functional (aka quality) requirements. Functional requirements would define, for example, what content should be displayed through HMI-rendering, or constraints on the

functionality emerging from the hardware platform architecture on the functionality (e.g. not exceeding the available memory or not misusing graphical processors pipelines). Typical examples of non-functional requirements are constraints on manufacturing costs or execution time. A notoriously difficult problem is to establish, at an early stage of development, whether satisfying such a set of requirements is feasible and what is the best design able to do so. This requires either to build numerous prototypes or to have a massive knowledge about each possible design, which is unrealistic in competitive markets where companies must deliver better solutions at a faster pace than their competitors.

In the domain of HMI-rendering for automotive, the embedded system consists of (1) a data processing application (i.e., a data-flow oriented system) and (2) resource-constrained hardware platform (i.e., heterogeneous hardware components like non-programmable processors and data storage units). This modular separation of concerns between the application (“*what we want to do*”) and the platform (“*what we can do*”) is typical in domains where execution platforms have to be considered to fulfil the requirements, such as the Internet of Things, embedded system, grid computing and even production plans.

The application and the platform are, however, not completely fixed as they have variation points. There are three main sources of variability. First, at the application level, multiple data-flow variants can achieve the functional requirements, differing in, e.g., the size of the flowing data chunks, the ordering of the operation tasks. Second, there exists a diversity of configurable hardware platforms which can differ in, e.g., memory capacities and processing pipelines. Third, there are various ways of mapping and deploying a given application on a specific platform, e.g., choose a processor to perform a given task or select a memory unit to store a given data.

This threefold variability is typical in automotive and many other kinds of embedded systems. Unfortunately, it leads to a high number of variants (1,548,288 in an industrial case we studied), each of which represents a specific *embedded system design alternative* (or *design* for short), that is, a specific mapping of a specific application variant to a specific platform variant. Among these design alternatives, not all are able to realize the functional requirements to the same extent, and the same holds for the non-functional requirements. Given the sheer number of variants, systematic consideration of all design alternatives is unfeasible for the software and system engineers. Efficient automations, therefore, appear as a necessity.

In this tutorial, we present a model-driven method we developed to tackle this problem. Our framework allows engineers to specify their embedded system in the form of variable dataflows and configurable hardware platforms. Then, a black-box tool we implemented can automatically derive and evaluate all potential designs (i.e. the compatible pairs of application and platform variants) with respect to all functional and quality requirements, as well as looking for the optimal design.

# Feature-Based Systems and Software Product Line Engineering: PLE for the Enterprise

Charles W. Krueger, Paul C. Clements

BigLever Software, Inc.

{ckrueger, pclements}@bigelever.com

## ABSTRACT

This paper describes a tutorial to introduce a product line engineering solution, including tools and methods, that is the subject of an upcoming ISO standard and known as “Feature-Based Systems and Software Product Line Engineering.” This tutorial will explain the approach, give its history and a brief summary of some of its many successes, and discuss its application to systems and software engineering. Moreover, the tutorial will cover how its usage is spreading beyond the traditional engineering realm, across the entire enterprise in areas such as product marketing, portfolio planning, manufacturing, supply chain management, product service and maintenance, and much more.

## CCS CONCEPTS

- Software and its engineering → Software product lines;

## KEYWORDS

Product line engineering, software product lines, feature modeling, bill-of-features, product portfolio, Feature-based PLE

## ACM Reference format:

Charles Krueger, Paul Clements, Enterprise Feature-Based Systems and Software Product Line Engineering: PLE for the Enterprise. In Proceedings of SPLC '19, Paris, September 9-13, 2019.  
<https://doi.org/10.1145/3336294.3342381>

## 1 Tutorial Topic

**Title:** Feature-Based Product Line Engineering for the Enterprise

**Goals:** To enable the audience to understand the importance and significance of Feature-Based Systems and Software Product Line Engineering (“Feature-Based PLE”); to learn the most important concepts of Feature-Based PLE; and to understand how its scope can extend across all of engineering and across an entire enterprise;

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

SPLC '19, September 9–13, 2019, Paris, France  
© 2019 Copyright is held by the owner/author(s).  
ACM ISBN 978-1-4503-7138-4/19/09  
<https://doi.org/10.1145/3336294.3342381>

**Intended audience:** The target audience for this tutorial includes industrial practitioners who are interested in the latest experiences with most efficient, effective and proven methods for transitioning to and sustaining software product line practice; and members of the research community who are interested in the new methods emerging from proven industry successes.

## 2 Tutorial Plan

This half-day tutorial’s topics include:

- introduction to Feature-Based Systems and Software Product Line Engineering and the PLE factory, which is the concept underlying Feature-Based PLE
- the use of *features* as the lingua franca for expressing product differences in all phases of the engineering and operations lifecycles, and a four-level structured ontology of features that, at each level, reduces the overall variation complexity of a product line and provides a view of the product appropriate for any level of an overall enterprise;
- the economics of Feature-Based PLE, as well as the organizational and business aspects of Feature-based PLE; proven approaches for adoption and institutionalization of PLE, including organizational change concepts that help organizations adopt PLE quickly and effectively;
- an update on the upcoming codification of Feature-Based PLE in an ISO standard.

**Justification of the tutorial for SPLC 2019:** Through this tutorial, SPLC attendees will be exposed to the highest level of PLE practice in the industry today, backed up with numerous industrial-strength real-world case studies of its application.

## 2 Presenters’ Backgrounds

**Dr. Charles Krueger** is the founder and CEO of BigLever Software, the long-standing leader in Product Line Engineering. With more than 30 years of experience in systems and software engineering practice.

**Dr. Paul Clements** is the Vice President of Customer Success of BigLever Software. He was previously a senior member of the technical staff at The Carnegie Mellon Software Engineering Institute (SEI) where he co-authored the book *Software Product Lines: Practices and Patterns*.

# Variability Modeling and Management of MATLAB/Simulink Models

Aitor Arrieta  
Mondragon University  
arrieta@mondragon.edu

## ABSTRACT

MATLAB/Simulink models are widely used in industry to model and simulate complex systems in several domains (e.g., automotive). These complex systems are produced in mass, and often, clients demand different functionalities. As a result, the variability of these models needs to be often considered. This tutorial aims at showing different variability modeling alternatives for MATLAB/Simulink users.

## CCS CONCEPTS

- Software and its engineering → Software product lines.

## KEYWORDS

MATLAB/Simulink, Variability modeling, Cyber-Physical Systems

### ACM Reference Format:

Aitor Arrieta. 2019. Variability Modeling and Management of MATLAB/Simulink Models. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/3336294.3342380>

## 1 INTRODUCTION

Nowadays, complex systems such as automotive systems or Cyber-Physical Systems are usually modeled and simulated using MATLAB/Simulink, as it allows to easily integrate complex physical dynamics models with software. These systems are evolving to be highly configurable to respond to client demands. MATLAB/Simulink offers some variability mechanisms (e.g., the variant subsystem, etc.). In addition, other mechanisms are also possible to model variability in such systems (e.g., uniform, negative or first class variability modeling [1]).

The main goal of this tutorial is to introduce MATLAB/Simulink users the different variability modeling approaches on Simulink models. The intended audience will mainly be practitioners, although researchers interested in variability modeling of Cyber-Physical Systems and similar systems might also be interested. To this end, the different MATLAB/Simulink variability modeling approaches will be reviewed and shown by means of illustrative examples.

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

*SPLC '19, September 9–13, 2019, Paris, France*

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7138-4/19/09.

<https://doi.org/10.1145/3336294.3342380>

## 2 PLAN

The length of the tutorial is estimated to be of half-day. This is the plan we intend to follow:

- Introduction to MATLAB/Simulink models
- Introduction to variability management with feature models
- Variability modeling alternatives in MATLAB/Simulink
  - Variant subsystems in MATLAB/Simulink
  - Uniform variability in MATLAB/Simulink
  - Negative variability in MATLAB/Simulink
- Hands-on: practical example(s)

Firstly, the tutorial will begin with a small introduction of MATLAB/Simulink models. Secondly, an overview of variability management with FeatureIDE will be given to ensure that everyone in the audience understands the general concepts of variability modeling. FeatureIDE is used for two reasons: (1) it is an open-source tool that can be installed by everyone and (2) the presenter has different implementations in MATLAB that helps on managing variability. Third, we will give an overview of the different alternatives to modeling variability in MATLAB/Simulink models and which is the most appropriate one depending on the user's needs. Lastly, there will be given the opportunity to follow, by employing one or two examples, the implementation of an entire product line in Simulink. The tutorial will have a first theoretical part with some small examples where the attendees will have the possibility of asking questions and “playing” with the tools, and a practical part with a bigger case study (one of the used in [2]). Thus, attendees are asked to bring their laptop along with a MATLAB version installed.

Examples and presentation will be available at the presenter's github account: <https://github.com/aitorarrietamarcos>

## REFERENCES

- [1] Aitor Arrieta, Goiuria Sagardui, and Leire Etxeberria. A comparative on variability modelling and management approach in simulink for embedded systems. *V Jornadas de Computación Empotrada, ser. JCE*, (26–33), 2014.
- [2] Aitor Arrieta, Shuai Wang, Goiuria Sagardui, and Leire Etxeberria. Search-based test case prioritization for simulation-based testing of cyber-physical system product lines. *Journal of Systems and Software*, 149:1–34, 2019.