SPLC Paris 2019
23rd Systems and Software Product Line Conference

23rd International Systems and Software Product Line Conference

Proceedings - Volume B

EDITED BY:

Carlos Cetina, Oscar Díaz, Laurence Duchien, Marianne Huchard, Rick Rabiser, Camille Salinesi, Christoph Seidl, Xhevahire Tërnava, Leopoldo Teixeira, Thomas Thüm, Tewfik Ziadi

SORBONNE UNIVERSITÉ
UNIVERSITÉ PARIS 1 PANTHÉON SORBONNE
ICPS Published by ACM

23rd International Systems and Software Product Line Conference

Proceedings - Volume B

Gold Sponsors

Supporters

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

ACM ISBN: 978-1-4503-6668-7

Table of Contents

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 Tërnava, *Sorbonne University, France*

Student Volunteers Chair

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

Finance Chair

Thùy Dodo, *Sorbonne University, France*

Program Committees

Demonstrations and Tools Track

Mathieu Acher	University of Rennes 1, France
Vander Alves	University of Brasília, Brazil
Ebrahim Bagheri	Ryerson University, Canada
Maurice H. ter Beek	ISTI–CNR, Italy
Elder Cirilo	Federal University of Sao Joao del-Rei, Brazil
Jane Cleland-Huang	University of Notre Dame, USA
Philippe Collet	Université Côte d'Azur, France
Holger Eichelberger	University of Hildesheim, Germany
Wolfram Fenske	Otto von Guericke University Magdeburg, Germany
Jose A. Galindo	University of Seville, Spain
Jianmei Guo	Alibaba Group, China
Jaejoon Lee	Lancaster University, United Kingdom
Lukas Linsbauer	Johannes Kepler University Linz, Austria
Jabier Martinez	Tecnalia, Spain
Natsuko Noda	Shibaura Institute of Technology, Japan
Gilles Perrouin	University of Namur, Belgium
Clément Quinton	University Lille, France
Marcello La Rosa	Queensland University of Technology, Australia
Christoph Seidl	TU Braunschweig, Germany
Ştefan Stănciulescu	ABB Corporate Research, Switzerland

VariVolution 2019: Second International Workshop on Variability and Evolution of Software-Intensive Systems

Workshop Organizers

Michael Nieke	TU Braunschweig, Germany
Jacob Krüger	University of Magdeburg, Germany
Lukas Linsbauer	Johannes Kepler University Linz, Austria
Thomas Leich	Harz University, Germany

Program Committee

Vander Alves	University of Brasília, Brazil
Sofia Ananieva	FZI Research Center, Germany
Jessie Carbonnel	University of Montpellier, France
Loek Cleophas	TU Eindhoven / Stellenbosch University, Netherlands/South Africa
Christoph Elsner	Siemens AG, Germany
Sandra Greiner	University of Bayreuth, Germany
Timo Kehrer	Humboldt-Universität zu Berlin, Germany
Juliana Alves Pereira	University of Rennes 1, France
Gabriela Sampaio	Imperial College London, England
Klaus Schmid	University of Hildesheim, Germany
Felix Schwägerl	MID GmbH, Germany
Stefan Sobernig	Vienna University of Economics, Austria
Ştefan Stănciulescu	ABB Corporate Research, Switzerland
Leopoldo Teixeira	Federal University of Pernambuco, Brazil
Mahsa Varshosaz	Halmstad University, Sweden
Manuel Wimmer	TU Wien, Austria

Steering Committee

Thorsten Berger	Chalmers | University of Gothenburg, Sweden
Timo Kehrer	Humboldt-Universität zu Berlin, Germany
Klaus Schmid	University of Hildesheim, Germany

WEESR 2019: Second International Workshop on Experiences and Empirical Studies on Software Reuse

Workshop Organizers

Jaime Chavarriaga	Universidad de Los Andes, Colombia
Julio Ariel Hurtado	Universidad del Cauca, Colombia

Program Committee

José Barros	Universidad de Vigo, Spain
María Cecilia Bastarrica	Universidad de Chile, Chile
David Benavides	University of Seville, Spain
César A. Collazos	Universidad del Cauca, Colombia
Elena Epure	Deezer, France
Héctor Florez	Universidad Distrital Francisco José de Caldas, Colombia
Thomas Fodgal	Danfoss Power Electronics A/S, Denmark

José Galindo	University of Sevilla, Spain
Jose García-Alonso	Universidad de Extremadura, Spain
Carlos Arce Lopera	Universidad ICESI, Colombia
Jabier Martinez	Tecnalia, Spain
Alicia Mon	Universidad Nacional de la Matanza, Argentina
María Constanza Pabón	Pontificia Universidad Javeriana, Colombia
Patricia Paderewski	Universidad de Granada, Spain
Daniel Perovich	Universidad de Chile, Chile
Rick Rabiser	Johannes Kepler University Linz, Austria
Francisco Álvarez Rodríguez	Universidad Autónoma de Aguascalientes, Mexico
Juan Manuel Murillo Rodríguez	University of Extremadura, Spain
Pedro Rossel	Universidad Católica de la Santísima Concepción, Chile

SPLTea 2019: Fourth International Workshop on Software Product Line Teaching

Workshop Organizers

Mathieu Acher	University of Rennes 1, France
Roberto E. Lopez-Herrejon	Université du Québec, Canada
Rick Rabiser	Johannes Kepler University Linz, Austria

Program Committee

Eduardo Almeida	Federal University of Bahia, Brazil
Don Batory	University of Texas at Austin, USA
Martin Becker	Fraunhofer IESE, Germany
Philippe Collet	Université Côte d'Azur, France
José A. Galindo	University of Seville, Spain
Jörg Kienzle	McGill University, Canada
Tomoji Kishi	Waseda University, Japan
Jaejoon Lee	Lancaster University, United Kingdom
Tomi Männistö	University of Helsinki, Finland
Sebastien Mosser	Universite du Quebec, Canada
Natsuko Noda	Shibaura Institute of Technology, Japan
Joost Noppen	BT Research and Innovation, England
Gilles Perrouin	University of Namur, Belgium
Clément Quinton	University Lille, France
Iris Reinhartz-Berger	University of Haifa, Israel
Andreas Wortmann	RWTH Aachen University, Germany

MODEVAR 2019: First International Workshop on Languages for Modelling Variability

Workshop Organizers

David Benavides	University of Seville, Spain
Rick Rabiser	Johannes Kepler University Linz, Austria
Don Batory	University of Texas at Austin, USA
Mathieu Acher	University of Rennes 1, France

Program Committee

Mathieu Acher	University of Rennes 1, France
Don Batory	University of Texas at Austin, USA
Maurice H. ter Beek	ISTI–CNR, Italy
David Benavides	University of Seville, Spain
Thorsten Berger	Chalmers | University of Gothenburg, Sweden
Philippe Collet	Université Côte d'Azur, France
Oscar Díaz	University of Basque Country, Spain
Lidia Fuentes	University of Málaga, Spain
José A. Galindo	University of Seville, Spain
Paul Gazillo	University of Central Florida, USA
Oystein Haugen	Østfold University College, Norway
Patrick Heymans	University of Namur, Belgium
Kyo Kang	Postech, Korea
Roberto E. Lopez-Herrejon	Université du Québec, Canada
Klaus Pohl	University of Duisburg-Essen, Germany
Rick Rabiser	Johannes Kepler University Linz, Austria
Camille Salinesi	University Paris 1 Panthéon-Sorbonne, France
Ina Schaefer	Technische Universität Braunschweig, Germany
Klaus Schmid	University of Hildesheim, Germany
Christoph Seidl	TU Braunschweig, Germany
Thomas Thüm	TU Braunschweig, Germany
Jules White	Vanderbilt University, USA
Tewfik Ziadi	Sorbonne University, France

REVE 2019: Seventh International Workshop on Reverse Variability Engineering

Workshop Organizers

Mathieu Acher	University of Rennes 1, France

Tewfik Ziadi	Sorbonne University, France
Roberto E. Lopez-Herrejon	Université du Québec, Canada
Jabier Martinez	Tecnalia, Spain

Program Committee

Aitor Arrieta	Mondragon University, Spain
Maurice H. ter Beek	ISTI–CNR, Italy
Danilo Beuche	pure-systems GmbH, Germany
Carlos Cetina	San Jorge University, Spain
Maxime Cordy	University of Luxembourg, Luxembourg
Oscar Díaz	University of Basque Country, Spain
João Bosco Ferreira Filho	Federal University of Ceara, Brazil
Stefan Fischer	Johannes Kepler University Linz, Austria
Jaime Font	University of San Jorge, Spain
José Galindo	University of Sevilla, Spain
Barbara Gallina	Mälardalen University, Sweden
Oystein Haugen	Østfold University College, Norway
Maritta Heisel	University of Duisburg-Essen, Germany
Sebastian Herold	Karlstad University, Sweden
Marianne Huchard	Université of Montpellier, France
Djamel Eddine Khelladi	University of Rennes 1, France
Jens Krinke	University College London, England
Axel Legay	UC Louvain, Belgium
Letitia W. Li	BAE Systems, USA
Lukas Linsbauer	Johannes Kepler University Linz, Austria
Leticia Montalvillo	IK4-IKERLAN Research Center, Spain
Jennifer Perez	Universidad Politécnica de Madrid, Spain
Gilles Perrouin	University of Namur, Belgium
Jacques Robin	University Paris 1 Panthéon-Sorbonne, France
Julia Rubin	University of British Columbia, Canada
Uwe Ryssel	pure-systems GmbH, Germany
Klaus Schmid	University of Hildesheim, Germany
Abdelhak-Djamel Seriai	University of Montpellier, France

Doctoral Symposium Track

Don Batory	University of Texas at Austin, USA
Mireille Blay-Fornarino	Université Côte d'Azur, France
Nicole Levy	Cedric, CNAM, France
Roberto E. Lopez-Herrejon	Université du Québec, Canada
Gilles Perrouin	University of Namur, Belgium

Visualization of Feature Locations with the Tool FeatureDashboard

Sina Entekhabi*
Middle East Technical University
Ankara, Turkey

Anton Solback
Chalmers University of Technology
Gothenburg, Sweden

Jan-Philipp Steghöfer
Chalmers | University of Gothenburg
Gothenburg, Sweden

Thorsten Berger
Chalmers | University of Gothenburg
Gothenburg, Sweden

ABSTRACT

Modern development processes and issue trackers often use the notion of features to manage a software system. Features allow communicating system characteristics across stakeholders and keeping an overview understanding—especially important for systems that exist in many different variants. However, maintaining, evolving or reusing features (e.g., propagating across variants, or integrating into a platform) requires knowing their locations to prevent extensive feature-location recovery. We advocate the use of embedded annotations, added directly into software assets by the developers during development. To support this process and provide immediate benefits to developers when using such annotations, we present the open-source tool FeatureDashboard. It extracts and visualizes features and their locations using different views and metrics. As such, it encourages developers recording features and their locations early, to prevent feature identification and location efforts, as well as it supports system comprehension.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*; *Software product lines.*

ACM Reference Format:
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 - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3307630.3342392

1 INTRODUCTION

Features are typically used to describe the functional and non-functional characteristics of a software system. Modern agile development as well as software product-line engineering (SPLE) processes use features to manage single or variant-rich systems. Features are units of communication, reuse, maintenance, and evolution, and they allow keeping an overview understanding of a system [5–7].

Effectively using features requires knowing their locations within the source assets of a software system. When not recorded, this information needs to be recovered, which is a daunting and error-prone task [15]—despite being one of the most common activities of developers. Notably, the locations of features are often scattered across the codebase [18, 19], further complicating this task. While automated techniques to recover feature locations have been proposed, none of them has found industrial adoption, mainly due to their low accuracy [9, 20], challenging their application in practice.

Using features is especially important in variant-rich systems. Organizations often realize variants through clone&own [8, 10] and propagate features across the cloned variants. When the number of variants challenges system evolution and maintenance, organizations often migrate the cloned variants into an integrated platform, which requires identifying features and their locations, and organizing the features in a feature model.

We take a different route and advocate the use of features early, and embedding feature locations directly into software assets. We rely on a lightweight annotation technique we developed before [2, 12], comprising in-code annotations, file and folder mappings, and a textual feature model in Clafer [3] syntax. Adding annotations during development is cheap, and they naturally co-evolve with the code (reducing annotation maintenance). They were also shown to be beneficial for variant maintenance and platform migration [12] as well as program comprehension [14]. However, standard code editors lack support for such annotations, while it is difficult for developers to extract feature information from large codebases manually. Consequently, using such annotations requires tool support that can extract and visualize the developer-defined features and annotations, supporting browsing and utilizing features.

We present FeatureDashboard,[1] a lightweight tool usable as an Eclipse plugin or standalone program that extracts features from the feature model as well as feature-to-code mappings and embedded annotations, and visualizes these using different views, as demonstrated below. We provide a demo video on the project's website,[2] showing a walk-through of FeatureDashboard's views.

2 TOOL OVERVIEW

We assume that a developer continuously documents features in a textual feature model (created in the project's root folder) and

*Sina Entekhabi contributed to the tool during an internship at Chalmers University of Technology, Sweden.

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342392

[1] https://bitbucket.org/easelab/featuredashboard
[2] https://bitbucket.org/easelab/featuredashboard/wiki/Demo

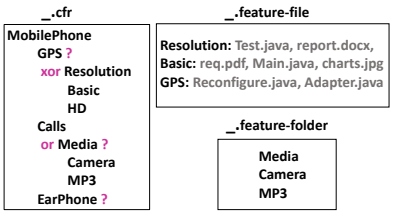


Figure 1: Examples of a textual (Clafer) feature model and mapping files, relating features to files and folders, used in FeatureDashboard

feature locations using in-code annotations as well as file and folder mapping files. The latter allow relating entire files or folders to a set of features (cf. Section 3). We encourage developers to perform this documentation continuously and immediately, when the feature information is still fresh in the developer's mind. To utilize FeatureDashboard's views, the developer selects the desired project in Eclipse's *Project Explorer*, and then opens the FeatureDashboard view. This view is the entry point for the tool, and it allows scanning the selected project's source tree for the feature model, the mapping files, and the in-file annotations. Once the scan is complete, the FeatureDashboard view represents all features in the project as extracted in the project's feature model, the annotated files, and the mapping files. By selecting features in this view and opening other, dedicated views of FeatureDashboard, the features are visualized for the developer. The developer can synchronise FeatureDashboard's view at any time when changes have been made in the project.

Feature location extraction and its visualization simplify common tasks such as change impact analysis where a developer needs to understand which parts of the code need to be changed when a feature is changed. It is also useful when introducing new features to understand the relationship to existing ones and to get a better understanding of the features, where they are implemented, and how they are tangled. FeatureDashboard can thus be used during the entire development lifecycle. FeatureDashboard's views can be classified into graphs and metrics views, and the majority work on one specific project. FeatureDashboard also offers a view to compare features across projects, which helps to detect inconsistencies (e.g., in the naming of features across projects) or to compare projects based on features.

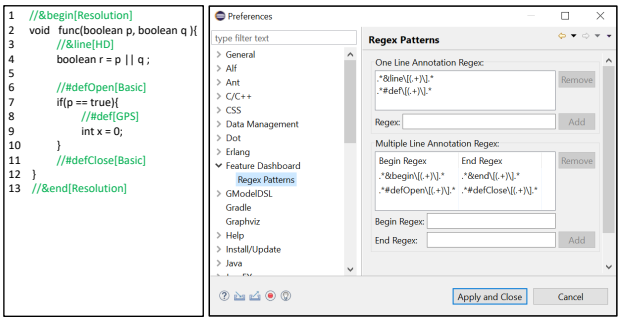


Figure 2: Java code with embedded annotations, whose syntax is customizable with regular expressions

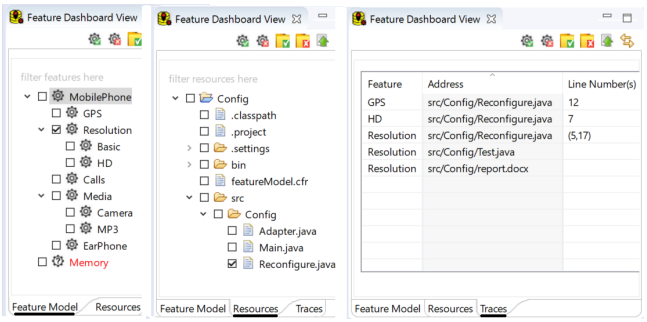


Figure 3: FeatureDashboard view

3 FEATURE DOCUMENTATION

Features and their locations are specified by developers as follows.

Feature Model. We support feature models expressed in the Clafer syntax [3], having an arbitrary name with *cfr* extension, and stored in each project's root folder. Currently, only the first file found will be parsed; future tool versions might include the possibility to merge feature models. Figure 1 shows an example. In Clafer, each feature is represented by its name in a single line, with the hierarchy being represented by tab indentation. Feature constraints (e.g., dependencies) can also be expressed, but currently only for documentation reasons (e.g., for a future product-line migration), without any visualization.

Embedded Annotations. Relying on our annotation approach [12], two kinds of annotations exist: for relating a feature to a single line of a file, and for relating a feature to multiple consecutive lines. The annotations are always escaped through the respective language's *commenting* syntax, to avoid affecting any other tooling (e.g., editors or compilers). Single-line annotations contain the name of exactly one feature. To relate consecutive lines of file to a feature, a pair of begin and end annotations is used in FeatureDashboard, each of which containing the same feature name. In the current version of FeatureDashboard no multiple features supported in one annotation, but it can be easily extended to support that.

In the Preferences of FeatureDashboard, developers can customize the syntax of these in-file annotations with regular expressions. FeatureDashboard recognizes the feature name as the first capturing group. Consequently, each regular expression must contain a capturing group sub-expression like `(.+)`. Figure 2 shows an example with Java code.

Mapping Files. Whole files and folders are mapped to features via simple text files named with the extensions `.feature-folder` and `.feature-file` (to avoid Eclipse hiding them, we suggest using _ as the file name). A `.feature-folder` file is put into the respective folder and just contains the names of the features (per line) that are intended to be mapped to that folder. The `.feature-file` files are also stored in folders, but contain feature names, separated with a colon from the list of file names (separated with commas) mapped to the feature. Each line of this file maps a feature to some files which exist in the same folder. Figure 1 shows examples.

4 VIEWS

Feature Dashboard View. This view is the main entry point for working with FeatureDashboard. For the selected project it shows

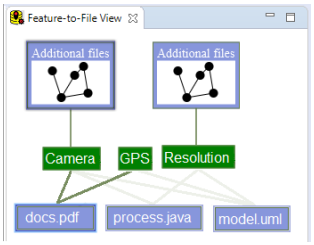


Figure 4: View for relationships of features to files

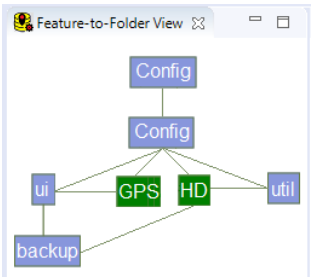


Figure 5: View for relationships of features to folders

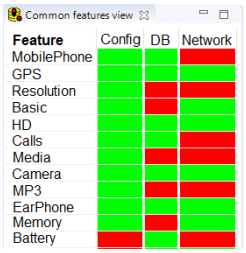


Figure 6: Common features between projects

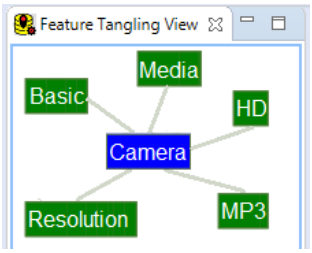


Figure 7: Features tangled with *Camera*

the feature hierarchy (from the model) and feature locations. Users can modify the preferences to filter files (e.g., specific file extension or folders). Three tabs are provided: *Feature Model*, *Resources*, and *Traces*. The features and the resources shown within the first two tabs are selectable, and the selections are used in the *Traces* tab.

The *Feature Model tab* shows the selected project's features along with their hierarchy according to the feature model defined in Clafer syntax in the project. The feature model representation of the model in Fig. 1 is shown in Fig. 3 within this tab. Here, features that are annotated in some source files of the project but not in the feature model are represented in red as root features. In this tab the features can be filtered and selected. Features selected in this tab are used by the tab *Traces* to show the selected features' locations and by other visualisation views, as explained in the following.

The *Resources tab* represents all files and folders in the imported project with their corresponding hierarchy, and allows filtering and selecting the resources for the user as shown in Fig. 3. Resources selected are used in the *Traces* tab to show the features related to those resources, and by other views of FeatureDashboard as well. Together with the *Feature Model* tab, it is thus possible to focus on relevant features and resources in all views of FeatureDashboard.

The *Traces tab* shows the feature locations of the selected features in the *Feature Model* tab and the selected resources in the *Resources* tab. Fig. 3 shows an example of feature locations in the Traces tab for the features and resources selected in the *Feature model* and *Resources* tabs. Each feature location is listed with its feature, the path within the project, and where applicable, the line numbers. If the relation of a feature and a file or folder is stated in a mapping file, the *line numbers* column is empty. The table of traces can be sorted and its data can be exported. In addition, double-clicking on a table row containing an embedded annotation will open the corresponding file and highlight the annotated line or code block.

Feature-to-File View. The relations between features selected in the FeatureDashboard view and relevant *files* are visualized in this view. Figure 4 shows an example. A connection between a feature (represented by a green rectangle) and a file (blue rectangle) indicates that the relation is established in a mapping file or by feature annotations within the file itself. It is also indicated if multiple selected features are implemented in the same files.

Double-clicking the *Additional files* box will show additional files that a feature is implemented in, but the other features are not. The reasoning behind this is to divide the information into different modules to reduce the elements in a single view. In this view, clicking on a file (`docs.pdf` in Fig. 4) will highlight the connections, which makes it easier to see which features are implemented in that file. Clicking on a file box will open that file in the editor and highlight the annotated code block.

Feature-to-Folder View. Similar to the *Feature-to-File* view, this view visualizes the relations between features and *folders*, as shown in Fig. 5. Green and blue rectangles represent features and folders, respectively. The links between a folder and a feature indicate that either the feature is annotated in a file within that folder or the relation is directly mentioned by a mapping file. Hierarchical folder representations are also shown by links between folders.

Common Features View. This view visualizes features different projects (or variants) have in common. A matrix that relates all open projects (rows) to all found features (columns) indicates that a feature is contained in a project with a green cell. An example is shown in Fig. 6.

Feature Tangling View. This view visualizes features that are tangled with each other. This is useful for identifying the parts of a system that are potentially affected if that particular feature is modified. It is also possible to see whether some selected features are tangled with each other in the *Feature-to-File* view. The *Feature Tangling* view, however, visualizes all tangled features for a selection. As with the other views, the considered feature must be selected in the *FeatureDashboard* view first. Figure 7 shows the tangled features with feature *Camera*. The blue rectangle in this view represents the selected feature, and the green ones are the features tangled with it. The link between the features can be double-clicked to see more detailed information regarding the two features in the *Feature-to-File* and *Feature-to-Folder* views.

Metrics View. All metrics proposed by Andam et al. [2] are calculated and shown in this view. The *Feature Metrics* tab shows feature metrics for the selected features in the selected projects, as shown in Fig. 8. The *Resource Metrics* tab shows feature-related metrics for different resources and averages for suitable metrics from the *Feature Metrics* tab. If the user has not selected any resources in

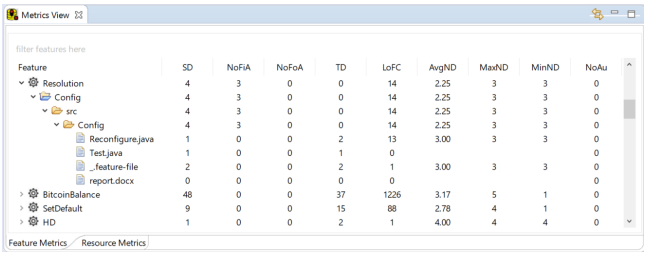


Figure 8: Metrics for features

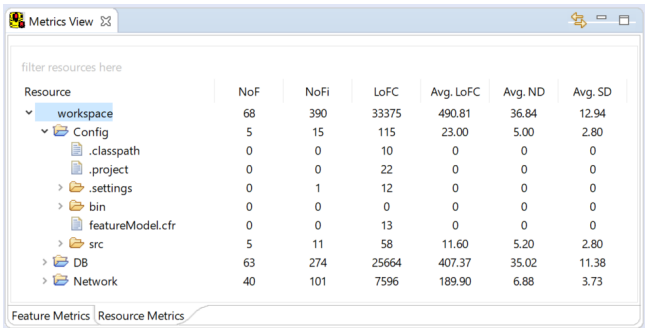


Figure 9: Metrics for resources

the *FeatureDashboard* view, metrics for the entire workspace are shown. Figure 9 demonstrates resource metrics in this tab.

History view. An experimental view was created to show how a selected metric for a feature evolved over time. Figure 10 shows how a feature's lines of feature code (LOFC) have change in successive commits based on the git history. This experimental feature will be developed further in the future.

5 RELATED WORK

The most closely related work is our previous tool FLOrIDA [2], from which FeatureDashboard gathers inspiration. However, FLOrIDA could not be released as open source and was a standalone Java Swing application, limiting integration into modern tools as well as extensibility. Both FeatureDashboard and FLOrIDA encourage developers to use embedded annotations [12], which can in the future by supported by a recommender system [1]. FeatureDashboard provides a superset of FLOrIDA's functionality, but does not have a built-in, automated feature-location tool (FLOrIDA has one based on the Lucence search engine).

Other works, such as CIDE [13], FeatureCommander [11], PEoPL [4, 17], and ViewInfinity [21], focus on visualizing variation points and optional features (omitting mandatory features, which are the focus of FeatureDashboard). They also focus on visualizing those variation points using different coloring techniques.

6 CONCLUSION

We described FeatureDashboard, an open-source tool to extract features and their locations in different artifacts, to calculate useful metrics for features, and to visualize the results. FeatureDashboard was designed to support developers recording feature information early and continuously, while obtaining immediate benefits through the different views that help browsing feature locations and keeping an overview understanding of the project. FeatureDashboard is open for extensions. We consider integrating it with Eclipse Capra[3] [16], a traceability management tool, which would allow managing and visualizing the traceability between features and many more types of assets, such as requirements, design models, and documentation with non-textual representations. In addition, we are currently evaluating the use of FeatureDashboard in industrial settings with our partners with a focus on the visualisations and the usefulness of the metrics.

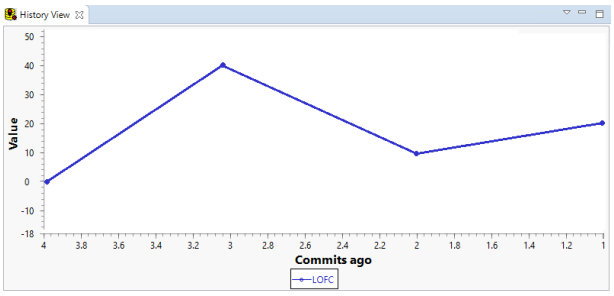


Figure 10: Experimental view showing a feature's history

REFERENCES

[1] Hadil Abukwaik, Andreas Burger, Berima Andam, and Thorsten Berger. 2018. Semi-Automated Feature Traceability with Embedded Annotations. In *ICSME*.
[2] Berima Andam, Andreas Burger, Thorsten Berger, and Michel RV Chaudron. 2017. Florida: Feature location dashboard for extracting and visualizing feature traces. In *VaMoS*.
[3] Kacper Bak, Zinovy Diskin, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2016. Clafer: unifying class and feature modeling. *Software & Systems Modeling* 15, 3 (2016), 811–845.
[4] B. Behringer, J. Palz, and T. Berger. 2017. PEoPL: Projectional Editing of Product Lines. In *ICSE*.
[5] 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*.
[6] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wasowski. 2014. Three Cases of Feature-Based Variability Modeling in Industry. In *MODELS*.
[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 *VaMoS*.
[8] John Businge, Openja Moses, Sarah Nadi, Engineer Bainomugisha, and Thorsten Berger. 2018. Clone-Based Variability Management in the Android Ecosystem. In *ICSME*.
[9] 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.
[10] 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*.
[11] Janet Feigenspan, Maria Papendieck, Christian Kästner, Mathias Frisch, and Raimund Dachselt. 2011. FeatureCommander: colorful# ifdef world.. In *SPLC Workshops*. 48.
[12] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining feature traceability with embedded annotations. In *SPLC*.
[13] Christian Kästner, Salvador Trujillo, and Sven Apel. 2008. Visualizing Software Product Line Variabilities in Source Code.. In *SPLC (2)*. 303–312.
[14] Jacob Krueger, Gul Calikli, Thorsten Berger, Thomas Leich, and Gunter Saake. 2019. Effects of Explicit Feature Traceability on Program Comprehension. In *FSE*.
[15] Jacob Krüger, Thorsten Berger, and Thomas Leich. 2018. *Features and How to Find Them: A Survey of Manual Feature Location.* Taylor & Francis Group, LLC/CRC Press.
[16] Salome Maro and Jan-Philipp Steghöfer. 2016. Capra: A Configurable and Extendable Traceability Management Tool. In *RE*.
[17] Mukelabai Mukelabai, Benjamin Behringer, Moritz Fey, Jochen Palz, Jacob Krüger, and Thorsten Berger. 2018. Multi-View Editing of Software Product Lines with PEoPL. In *40th International Conference on Software Engineering (ICSE), Demonstrations Track*.
[18] Leonardo Passos, Jesus 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*.
[19] 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). Preprint.
[20] Julia Rubin and Marsha Chechik. 2013. A survey of feature location techniques. In *Domain Engineering*.
[21] Michael Stengel, Mathias Frisch, Sven Apel, Janet Feigenspan, Christian Kastner, and Raimund Dachselt. 2011. View infinity: a zoomable interface for feature-oriented software development. In *ICSE*.

[3] https://eclipse.org/capra

symfinder: A Toolchain for the Identification and Visualization of Object-Oriented Variability Implementations

Johann Mortara
johann.mortara@univ-cotedazur.fr
Université Côte d'Azur, CNRS, I3S,
Sophia Antipolis, France

Xhevahire Tërnava
xhevahire.ternava@lip6.fr
Sorbonne Université, UPMC, LIP6,
Paris, France

Philippe Collet
philippe.collet@univ-cotedazur.fr
Université Côte d'Azur, CNRS, I3S,
Sophia Antipolis, France

ABSTRACT

When variability is implemented into a single variability-rich system with object-oriented techniques (*e.g.*, inheritance, overloading, design patterns), the variation points and variants usually do not align with the domain features. It is then very hard and time consuming to manually identify these variation points to manage variability at the implementation level. *symfinder* is a toolchain to automatically identify and visualize these variability implementation locations inside a single object-oriented code base. For the identification part, it relies on the notion of symmetry between classes or methods to characterize uniformly some implementation techniques such as inheritance, overloading, or design patterns like Factory. The toolchain also generates an interactive Web-based visualization in which classes that are variation points are nodes linked together through their inheritance relationships, while the size, color, and texture of the nodes are used to represent some metrics on the number of overloaded constructors or methods. As a result, the visualization enables one to discern zones of interest where variation points are strongly present and to get relevant information over concerned classes. The toolchain, publicly available with its source code and an online demo, has been applied to several large open source projects.

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:
Johann Mortara, Xhevahire Tërnava, and Philippe Collet. 2019. symfinder: A Toolchain for the Identification and Visualization of Object-Oriented Variability Implementations. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3307630.3342394

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342394

1 INTRODUCTION

At the domain level of a Software Product Line (SPL), variability of products is usually captured through common and variable domain features, these features being realized in software assets, such as code, at the implementation level. From the many existing variability implementation techniques, annotation-based ones, such as condition compilation with preprocessors [12], have been heavily used in embedded systems, but they tend to pollute code with a low abstraction level [11] and to produce errors at derivation time [9]. Furthermore, feature modules is a specific technique that structures all lines of code corresponding to a domain feature into a coding unit called feature [2]. While it looks preferable, it does not support well cross-cutting variability [8], and it implies code refactoring that may be completely unfeasible in practice for many systems. This is typically the case in variability-rich systems that have progressively introduced variability into object-oriented code, using many different traditional techniques, such as inheritance, overloading, and design patterns [4, 17]. Variability implementations then do not align well with domain features and identifying where these implementations are precisely located is crucial to manage this kind of variability [15].

While approaches and techniques have been proposed to partially locate domain features at the code level [3, 16], there is no work dealing with the identification of object-oriented variability implementations at the structural level, namely at the level of variation points (*vp*-s) and variants [14, 17]. Contrary to a feature related to the variability domain, a variation point represents one or more locations in code at which variation will occur, while the way that a variation point is going to vary is defined by its variants [7].

This paper introduces *symfinder*, a toolchain to automatically identify and visualize these variability implementation locations inside the code base of a single variability-rich Java system. For the identification part, it relies on the notion of symmetry (defined in Section 2) [5, 20] between classes or methods to characterize uniformly variation points and variants from several implementation techniques, such as inheritance, overloading, or design patterns. Identified variation points and variants are stored into a graph database and reused to generate an interactive web-based visualization. This visualization enables one to discern zones of interest where variation points are strongly present and to get relevant information over concerned classes. Details about the usage of symmetry, the identification approach, and its validation over a set of large projects are available in [18].

External information about the *symfinder* toolchain to be demonstrated are as follows:

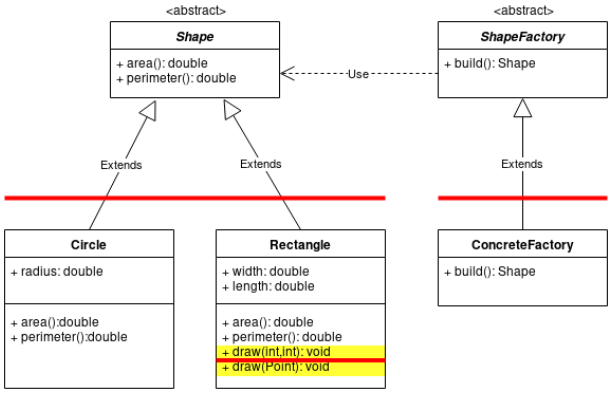


Figure 1: Example of variation points, as local symmetries (illustrated with red lines), in code asset representations

- *symfinder* source code is publicly available at https://github.com/DeathStar3/symfinder, including some usage guidelines;
- a video demonstrating the visualization part is available at https://www.youtube.com/watch?v=wb3U6MJ7nAM;
- experimental results from [18] and an online demo are available at https://deathstar3.github.io/symfinder-demo/.

In the remainder of this paper, we first give some background on variation points and symmetry in object-oriented code (Section 2). Next, we describe *symfinder*, giving details on the identification and visualization parts, as well as on the portability and performance of the toolchain (Section 3). We summarize current applications (Section 4) and finally, we conclude the paper by briefly discussing future work (Section 5).

2 VARIATION POINTS AND VARIANTS IN OBJECT-ORIENTED SYSTEMS

Let us consider an illustrative example with a Java implementation of a family of geometric shapes, such as rectangles and circles (*cf.* Figure 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 two ways for drawing shapes, namely the `draw()` method in `Rectangle`. Finally, the `ShapeFactory` and `ConcreteFactory` classes implement a Factory pattern, with an abstract `build()` method that is implemented in the subclass by returning an instance of a subclass of `Shape` depending on some configuration values.

As stated in the introduction, a variation point corresponds to a place in code where some variation happens, and the way this point vary corresponds to variants [7]. Therefore, the example in Figure 1 exhibits three variation points:

- the abstract class `Shape` is common, thus a variation point, for two variants `Rectangle` and `Circle`;
- the abstract creator class `ShapeFactory` is another variation point for its single variant `ConcreteFactory`;
- the method `draw()` in class `Rectangle` is another variation point for two overloaded variants that have different arity.

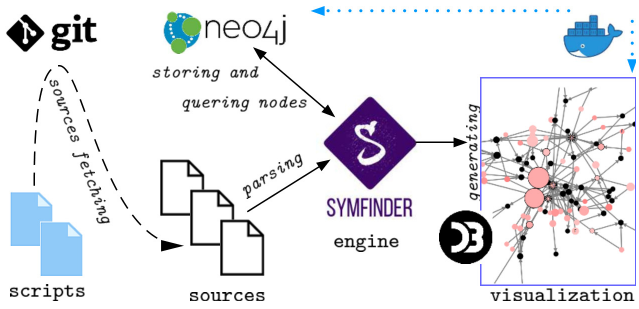


Figure 2: The dockerized *symfinder* toolchain

Symmetry and local symmetry have been recognized as a way to comprehend and create order in nature and human made artifacts [1]. They have also been studied in software and been identified in different constructs of programming languages or object-oriented design patterns [5, 19] where, in accordance with the symmetry definition, a part of their design remains *unchanged* while another may *change*. For example, subtyping in the `Shape` hierarchy in Figure 1 exhibits the property of symmetry. The classes of this type path, `Circle` and `Rectangle`, change while they preserve and conform to the common behavior defined in the superclass `Shape`.

A more complete study of the symmetry in object-oriented constructs can be found in the companion research paper [18], as well as the definition of the relationship between variation points and symmetry. Basically, as variation points and variants, respectively, mark the unchanged and changeable parts in a design, they are actually the local symmetric places in design. Hence, all the techniques used to implement the three variation points in Figure 1 uniformly exhibit the property of symmetry.

3 SYMFINDER

3.1 Overview

The aim of *symfinder* is to enable automatic identification and visualization of different local symmetries (*i.e.*, variation points) in a variability-rich Java-based system. The main purpose is to provide help in understanding the variability places of a variability-rich system during its development. Figure 2 depicts its whole toolchain, which consists of three parts. First, the sources of a targeted Java project are fetched from its *git* repository, then the *symfinder* engine enables the automatic identification of all its *vp*-s, through the property of local symmetries, and builds a graph representation of them, and finally a visualization of the identified *vp*-s is generated and can be navigated through a web browser.

3.2 Identification through local symmetries

For a targeted variability-rich system, local symmetries are identified according to the defined symmetry in each language construct, technique, and design pattern given in the companion research paper [18] and summarized 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,

Table 1: Six language features, their symmetries, and their respective visualization as nodes with their relationships

Language feature	Visual node	Commonality /Unchange	Variability /Change
Class as type		Class	Objects
Class subtyping		Superclass	Subclasses
Interface		Type	Implem. classes
Method overloading		Structure	Signatures
Constructor overloading		Structure	Signatures
Strategy Pattern	S	Strategy interface	Algorithms
Factory Pattern	F	Abstract Creator and product	Concrete creators and products
Inheritance			

including the concrete overloaded constructors and methods are also identified, representing their respective variants.

Technically, the *vp*-s identification process is achieved in two steps. First, the targeted Java system is parsed and the structure of its implementation units is stored into a Neo4j graph database where each class, interface, and method is represented by a node, including its structural relationships with other nodes. The node and relationship types are also labeled, for example, `CLASS` and/or `ABSTRACT` for nodes, `EXTENDS` or `IMPLEMENTS` for inheritance relationships. Secondly, *vp*-s are identified by querying the database for specific paths in the labeled graph using the Cypher language [1]. For example, the following query uses labels (*e.g.*, `CLASS`, `EXTENDS`, and `IMPLEMENTS`) to identify some *vp* and its variants:

```
MATCH (c)-[:EXTENDS|:IMPLEMENTS]->(c2:CLASS)
WHERE ID(c) = $id
RETURN count(c2)
```

When the `$id` of the node corresponds to the `Shape` class (*cf.* Figure 1), then the query will count the number of its `EXTENDS` relationships (*i.e.*, 2). Thus, `Shape` is identified as a *vp* with its two variants. Similarly, other queries are used to identify method level *vp*-s, as well as the strategy and factory design patterns. In that last case, more complex queries analyse respectively the structural relationship of classes, and the return types of methods, a detected factory being a class that contains a method returning an object whose type is a subtype of the declared method return type. Moreover, the four main queries used to count the number of *vp*-s and variants, at class and method level are documented here: https://github.com/DeathStar3/symfinder/blob/splc2019-artifact/detection_method.md.

3.3 Web-based visualization

The *symfinder* toolchain provides the capability to generate an interactive Web-based visualization of the identified *vp*-s (*cf.* Figure 2). After considering the visualization capabilities of Neo4j and other visualization forms used in SPL engineering [13], we decided to use the D3.js [2] library as the visualization support, so that only a Web browser is needed to visualize the *vp*-s graph.

[1] https://neo4j.com/developer/cypher/
[2] https://d3js.org/

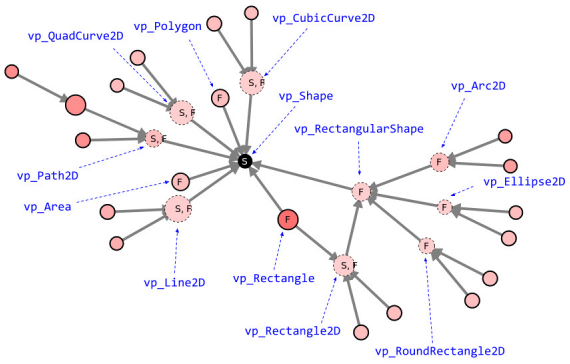


Figure 3: Excerpt of a visualization of identified *vp*-s in the Java AWT library. Annotations in blue show potential *vp*-s names that are displayed when hovering a node.

Instead of visualizing the graph of *vp*-s with variants by plain nodes and edges, we consider that it is important to also visualize information regarding the used language constructs, techniques, or design patterns for implementing variability. For this reason, as in many software and code artifacts visualizations [10], we rely on the visual principles of preattentive perception [6] 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 seven kinds of nodes that we use in *symfinder* for the visualization of the kinds of potential *vp*-s with variants are shown in Table 1.

As an example, Figure 3 shows a visualization excerpt of the identified *vp*-s in the Java AWT library. It shows 30 identified *vp*-s without variants, where each *vp* node is represented by a circle, and edges are class extension or interface implementation. Using Table 1, one can interpret that the black circle with letter `S` of `vp_Shape` denotes its relation to an interface that is also a Strategy pattern in code. In addition, the larger size of node `vp_Line2D` denotes that it contains variability at method level through overloading. Similarly, the more intense color of `vp_Rectangle` denotes that it contains variability at constructor level.

The visualization itself has five main features: (1) the visualized graph of *vp*-s can be zoomed in and out, (2) the *vp* label appears when hovering the node, (3) the out-of-scope *vp*-s [18] can be filtered out (*e.g.*, `java.awt.images` when analysing the variability of Java AWT library), (4) the isolated nodes can also be filtered out, and (5) the number of identified *vp*-s and variants.

3.4 Portability and performance

In order to achieve maximum portability, the execution of the *symfinder* toolchain is fully dockerized. Thus, only Docker [3] and Docker Compose [4] are required to run the toolchain. Internally, three Docker environments are executed sequentially, which correspond to the three parts of the toolchain enumerated in Section 3.1:

[3] https://www.docker.com/
[4] https://docs.docker.com/compose/overview/

Table 2: Execution time of all steps when analysing JFreeChart 1.5.0.

Step	Execution time [hh:mm:ss]
Detection of methods and classes	00:19:42.702
Creation of inheritance relationships	00:02:24.419
Detection of strategy patterns	00:01:22.376
Detection of factory patterns	00:02:19.115

(1) A Docker container that fetches sources and checks out the desired tags or commits of a variability-rich system from its *git* repository (*cf.* Figure 2). This enables *symfinder* to work easily over any Java system that is publicly available.
(2) The identification process is run by deploying a Docker Compose environment per project that is made of two Docker containers, the *symfinder* engine and an instance of a Neo4j database. Then, a *"runner"* Docker container automates the execution of *symfinder* on multiple systems.
(3) The visualization is provided by running a Docker container that runs a lightweight Python server exposing the generated HTML files on http://localhost:8181.

Finally, the deployment of the toolchain is made through *shell* scripts that encapsulate the Docker commands to run. *symfinder*'s deployment is validated on three operating systems, GNU/Linux, macOS Sierra 10.12 or newer on hardware from at least 2010, and Windows 10 64-bit (Pro, Enterprise or Education)[5].

Regarding its performance, a *symfinder* execution mainly depends on the size of the analysed system. For example, the execution time over the JFreeChart 1.5.0 (*cf.* Section 4) with 94,384 LoC takes around 26 minutes on a virtual machine with 1 core of Xeon E5-2637 at 3.50GHz and 128 GB of memory, to detect an overall number of 1415 variation points. Details on the time spent on each main step are given in Table 2. The detection of methods and classes is taking 75% of the time as it needs to traverse all classes and methods, while the other steps reuse the built graph.

4 CURRENT APPLICATIONS

We evaluated the identification and visualization of *vp*-s, including the portability and performance of *symfinder*, by conducting several experiments with realistic variability-rich systems. Specifically, we applied *symfinder* over eight open source systems, namely Java AWT, Apache CXF 3.2.7, JUnit 4.12, Apache Maven 3.6.0, JHipster 2.0.28, JFreeChart 1.5.0, JavaGeom, and ArgoUML. Their analysed tags, commits, size in LoC, and some basic metrics on the identified number of *vp*-s with variants are given in the companion research paper [18]. As the main result of these successful applications of *symfinder*, we were able to identify the first patterns of variability in object-oriented variability-rich systems. Still, all conducted experiments are available at https://deathstar3.github.io/symfinder-demo/, which are illustrated with extracted screenshots, explanations, and then a demonstration of their visualization is also deployed online. A specific set of experiments is also available as an online demo.

[5]Its deployment is not possible on Windows Home, which is a limitation of Docker.

5 CONCLUSION

symfinder is a toolchain that supports identification and visualization of different kinds of variation points with variants of a variability-rich Java system using the property of symmetry in object-oriented software constructs. The toolchain source code is publicly available and can be easily used through a containerized version on Docker.

Future work includes toolchain extensions to identify symmetries in other language features, being object-oriented or functional, integrating other properties from Alexander's theory of centers [1], and studying how to handle code evolution w.r.t. variation points identification and visualization.

REFERENCES

[1] 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.
[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.* Springer.
[3] Wesley KG 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] Rafael Capilla, Jan Bosch, and Kyo-Chul Kang. 2013. *Systems and Software Variability Management.* Springer.
[5] 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.
[6] Stephan Diehl. 2007. *Software visualization: visualizing the structure, behaviour, and evolution of software.* Springer Science & Business Media.
[7] Ivar Jacobson, Martin Griss, and Patrik Jonsson. 1997. *Software reuse: architecture, process and organization for business success.* ACM Press/Addison-Wesley Publishing Co.
[8] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A variability-aware module system. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 773–792.
[9] 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.
[10] Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin Pinzger. 2005. Codecrawler: an information visualization tool for program comprehension. In *Proceedings of the 27th international conference on Software engineering*. ACM, 672–673.
[11] 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.
[12] 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.
[13] 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.
[14] Angela Lozano. 2011. An overview of techniques for detecting software variability concepts in source code. In *International Conference on Conceptual Modeling*. Springer, 141–150.
[15] 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.
[16] Julia Rubin and Marsha Chechik. 2013. A survey of feature location techniques. In *Domain Engineering*. Springer, 29–58.
[17] Xhevahire Tërnava 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.
[18] Xhevahire Tërnava, Johann Mortara, and Philippe Collet. 2019. Identifying and Visualizing Variability in Object-Oriented Variability-Rich Systems. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*. ACM.
[19] Liping Zhao. 2008. Patterns, symmetry, and symmetry breaking. *Commun. ACM* 51, 3 (2008), 40–46.
[20] Liping Zhao and James Coplien. 2003. Understanding symmetry in object-oriented languages. *Journal of Object Technology* 2, 5 (2003), 123–134.

FlexiPLE - A Tool for Flexible Binding Times in Annotated Model-Based SPLs

Dennis Reuling
dreuling@informatik.uni-siegen.de
Software Engineering Group, University of Siegen, Germany

Christopher Pietsch
cpietsch@informatik.uni-siegen.de
Software Engineering Group, University of Siegen, Germany

Udo Kelter
kelter@informatik.uni-siegen.de
Software Engineering Group, University of Siegen, Germany

Manuel Ohrndorf
mohrndorf@informatik.uni-siegen.de
Software Engineering Group, University of Siegen, Germany

ABSTRACT

Annotative approaches are commonly used to specify variation points in a Model-based software product line (MBSPL) implementation. Variant-specific parts are marked using annotated presence conditions in a so-called 150% model. Such approaches lead to a static feature selection (or *binding*), which is appropriate, i.e., for platform-specific features. However, dynamic binding, i.e. feature selection at run time, is a necessity in many (industrial) contexts. We present our tool FlexiPLE which allows for a feature-wise binding time selection, even *after* a MBSPL implementation. To this end, our tool a) supports the definition of binding time constraints, b) aids the user in the adaption process based upon variability analysis and c) ensures that only valid feature combinations can be selected statically *and* dynamically by incorporating a staged configuration approach. Our tool builds upon a robust stack of state-of-the-art technologies and tools in the context of MBSPLs.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; **Software product lines**; *Software evolution*;

KEYWORDS

Model-based Software Product Line Engineering, Staged Configuration, Binding Times, Variability Encoding

ACM Reference Format:
Dennis Reuling, Christopher Pietsch, Udo Kelter, and Manuel Ohrndorf. 2019. FlexiPLE - A Tool for Flexible Binding Times in Annotated Model-Based SPLs. In *Proceedings of 23rd International Systems and Software Product Line Conference - Volume B, Paris, France, September 9–13, 2019 (SPLC '19),* 4 pages.
https://doi.org/10.1145/3307630.3342395

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342395

1 INTRODUCTION

A *software product line* (SPL) is a family of similar software systems which differ in certain *Variation Points* (VP)[2]. A member of an SPL is specified by its features, which imply a choice for all VPs. During product derivation, features can be bound either *statically* at compile or build time or *dynamically* at load or run time. Dynamic SPLs (DSPLs) have been proposed in [6]. The pros and cons of both binding times are analyzed in detail in [11]. It is shown that both are needed in general, which is in line with experiences made in industrial contexts [3, 7]. Tools supporting *Software Product Line Engineering* (SPLE) should enable developers to choose a binding time individually for each feature. To achieve this, the concept of a *Staged Configuration* is commonly employed [4, 6]. For SPLs implemented using conventional source programs, Rosenmüller et. al [11, 12] propose tools for feature-wise staged configuration. In domains such as automotive systems and embedded software, source code is frequently replaced by models, leading to Model-based SPLs (MBSPLs) [7, 13].

Existing methods and tools for staged configuration of source code-based SPLs cannot be applied to MBSPLs: they depend a lot on the supported programming language and its semantics. This dependency also applies to modeling languages used in MBSPLs, as will be shown by an example introduced in the next section. Section 3 details the functions of our tool FlexiPLE to support a staged configuration of MBSPLs. The implementation of FlexiPLE is sketched in Section 4. FlexiPLE, a tool demonstration as well as related documentation can be found at [1].

2 RUNNING EXAMPLE

Our running example, the *Pick and Place Unit (PPU)*, is a case study for embedded software in industrial automation [13]. Figure 1 depicts an excerpt of the *Feature Model (FM)*, which specifies some of the features and domain constraints of the PPU.

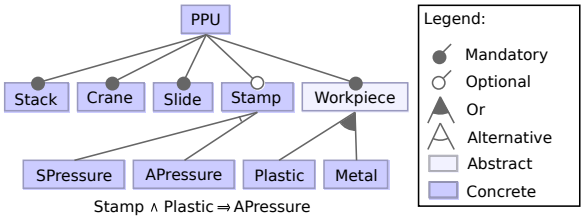


Figure 1: Feature Model of the PPU MBSPL

The mandatory *Stack* stores work pieces. A *work piece* can consist of *metal* or *plastic*. The mandatory *Crane* transports work pieces between the *Stack* and the output storage, a mandatory *Slide*. Optionally, a *Stamp* labels work pieces. There are two types of stamps: they label work pieces using a standard pressure (*SPressure*) or an adjusted pressure (*APressure*). The former one can only be used for metallic work pieces, the later one can also label plastic work pieces. Furthermore, when labeling only metallic work pieces, plastic work pieces are directly transported to the output storage.

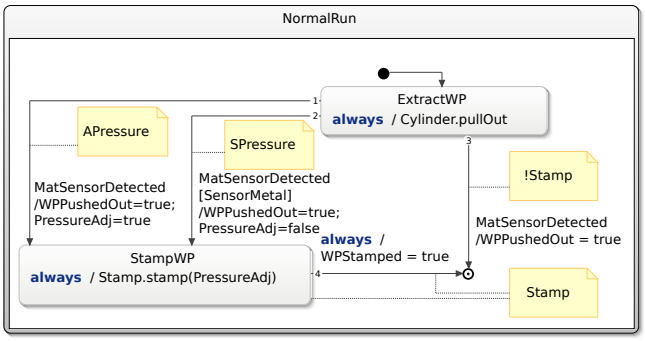


Figure 2: Annotated Statechart (150% Model)

Figure 2 shows an excerpt of the 150% statechart model. It describes the behavior of the *Stack* and its interaction with the *Stamp*. Initially, the cylinder of the hardware component is pulled out in order to extract a work piece from the input storage. If a work piece has been successfully extracted the event `MatSensorDetected` is called and depending on the bounded VPs the corresponding transition is triggered and variables are set accordingly (i.e., `WPPushedOut`). VPs are realized by presence conditions using annotations. For instance, transition 1 is annotated with *APressure*. In case of static binding, this transition is removed if a feature configuration excludes *APressure*, and a new product is generated and installed. Now, we assume that a customer wants to switch dynamically between either labeling only metallic or any type of work pieces, for example to increase the throughput if only metallic work pieces should be labeled. To achieve this flexibility after deployment of the PPU, we have to dynamically bind the features *APressure* and *SPressure*.

3 APPROACH OVERVIEW

Our tool supports the shifting from static to dynamic binding following a staged configuration process [3, 4, 6]. The tool pipeline is sketched in Fig. 3. The tool performs the following main steps:

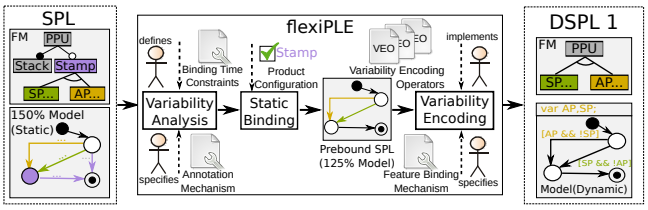


Figure 3: Overview of our Approach (using the PPU).

Variability Analysis. First, the VPs of the MBSPL are analyzed based upon *binding time constraints* and the *annotation mechanism* which is used in the 150% model. Here, we differentiate between:

Binding Time Analysis: Binding time constraints are defined by the developer. They specify whether features are bound statically or dynamically. FlexiPLE analyses each constraint for binding time dependencies among features for ensuring correctness during the staged configuration process, similarly to *dynamic binding units* as presented in [11].

If a binding time constraint for feature f specifies that this feature should be bound dynamically then all features a) that require feature f or b) that are in the same feature group must also be bound dynamically. In this case, the binding time constraint of f is propagated to the dependent features. In our running example, if feature *SPressure* shall be bound dynamically, the feature *APressure* is (automatically) defined as dynamic in order to offer at least one viable alternative if feature *SPressure* is deselected dynamically (i.e., at run time).

Variation Point Analysis: Based upon the (derived) binding time constraints and the specification of the *annotation mechanism* used in the 150% model (s. Sect. 4), FlexiPLE searches for annotated parts in the 150% model. Each found variation point (i.e., annotated model element) is classified according to the binding time of its presence condition. In our example statechart in Fig. 2, our tool finds five variation points (transitions 1–4 and State *StampWP*). Due to the dynamic features *APressure* and *SPressure*, the variation points belonging to transition 1 and 2 are classified as dynamic. The remaining variation points are classified as static (due to feature *Stamp*).

Static Binding. Based upon the results of the variability analysis as well as a (valid) static feature selection (see Fig. 3 center), our tool (pre-)binds all variation points. For each static variation point a) its model element(s) are removed if its presence condition evaluates to *false* and b) its annotations are deleted from the model. After having configured all static features, the resulting so-called (pre-bound) 125% model still contains all dynamic variation points. In our example (see Fig. 2), this step removes transition 3 and annotations *Stamp* and *!Stamp* because feature *Stamp* is selected. The remaining variation points of transitions 1 and 2 are left "as-is" due to their dynamic binding time.

Variability Encoding. In order to implement dynamic binding, we utilize the concept of variability encoding, which is known from SPLs [14] as well as code/model merging [9, 10]. We integrate the annotated variability information (i.e., presence conditions) using built-in language constructs. Such constructs are usually available in complex languages, mostly to support reuse among variants of model elements and/or semantic constructs [8, 10]. Thus we assume a catalog of variability encoding operators (VEO) to be available (or to be implemented in our tool, see Sect. 4). VEOs define how to adapt the 125% model of the specific modeling language, i.e., how to achieve dynamic binding.

For statecharts, as used in our example, one straightforward VEO is to encode annotated transitions using additional guard constraint(s). Applying this operator to our pre-bound SPL yields the guards [APressure] at transition 1 and [SPressure && SensorMetal] at transition 2 (see Fig. 2). Although this allows dynamic binding, invalid feature combinations can be selected dynamically. To ensure valid configurations during *all* binding times, FlexiPLE integrates additional constraints from the feature model. Each (to be encoded)

presence condition is analyzed regarding its relations to other dynamic features. For example, due to the exclusion relation between features *SPressure* and *APressure*, we enrich the (encoded) guard expression as presented in Fig. 3 (right) for ensuring their mutual exclusion. Finally, our tool employs a specified *Feature Binding Mechanism* to declare and initialize corresponding variables for each (newly) introduced guard expression term (see *var* header in the final DSPL in Fig. 3).

4 TOOL IMPLEMENTATION

We implemented FlexiPLE as plugin in the Eclipse Modeling ecosystem[1] and used a robust stack of MBSE and SPL technologies like *Viatra*[2] and *FeatureIDE*[3]. One main goal of our tool is to aid the user during the tool configuration process for ensuring that a) the manual development effort is reduced significantly and that b) all mandatory artifacts are available and (at least syntactically) correct. We achieve this using pre-generated stubs and well-defined interfaces used throughout FlexiPLE (s. screencast at [1]).

Components. An overview of all components as well as re-used and integrated external components is depicted in Fig. 4. The core component *Staged Composer* employs different components, each offering different functionalities. The *Dynamic FM Editor* component enables users to define binding time constraints. It is integrated in the widely-used feature model editor of *FeatureIDE*. The *Binding Time Analyzer* delegates the analysis to the integrated *FM Analyzer* for deriving binding times as described in Sect. 3. FlexiPLE allows for arbitrary annotation mechanisms used in the input 150% model, thus the developer has to specify the used annotation mechanism as model query pattern. Based upon this pattern, the *Variation Point Analysis* component uses the Viatra *Model Query Engine* and provides all found variation points to the *Static Composer* (for static VPs) as well as to the *Variability Encoder* (for dynamic VPs) based upon their (derived) binding time. Additionally, VEOs for *all* found dynamic variation points are generated in terms of model transformation rule stubs, thus achieving complete coverage for variability encoding. Using the provided static variation points and a valid product configuration the *Static Composer* then creates a 125% model, which is in turn used for variability encoding. This 125% model is transformed using the Viatra *Transformation Engine* based upon the VEOs as well as the feature binding mechanism provided as Java implementation. In our running example (see Fig. 3 right), the resulting dynamic statechart is the final output of the *Staged Composer* (and thus FlexiPLE). It is simulated using the *Statechart Simulator* as provided by Yakindu[4].

Tool Perspective. The tool perspective, which is shown in Fig. 5, integrates all (frontend and backend) components of FlexiPLE. More specifically, it consists of four main views (1 – 4):

- **Project Explorer** 1: This view enables the developer to inspect the current (Feature) project as known from *FeatureIDE*. Additionally, FlexiPLE offers pre-generated stubs and classes for implementing all necessary artifacts 5 (annotation mechanism, feature binding mechanism and VEOs).

[1]https://www.eclipse.org/modeling/
[2]https://www.eclipse.org/viatra/
[3]https://featureide.github.io/
[4]https://www.itemis.com/en/yakindu/state-machine/

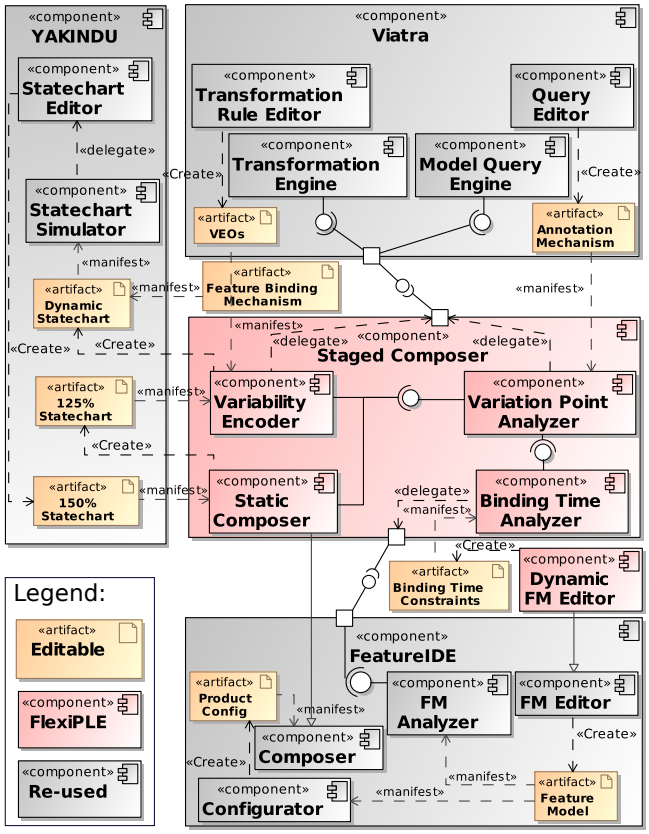


Figure 4: Tool Components used in FlexiPLE.

Due to annotated statechart transitions in case of dynamic features *SPressure* and *APressure*, FlexiPLE pre-generated VEO *Transition.xtend* which is to be implemented by the developer. The final (potentially dynamic) statechart model(s) are to be found in the *products* folder 6, as usual.

- **Extended Feature Model Editor** 2: FlexiPLE extends the default feature model editor by a) adding new binding time icons to the feature model and legend as well as b) new context menus for defining the binding time of each feature 7. After each edit, the binding time constraints are updated in the editor accordingly, including derived binding times.
- **Statechart Editor** 3: This editor allows for inspecting the final (encoded) statechart as well as its current state during simulation. The currently traversed transition as well as state is highlighted in yellow. Regarding our running example, one can see the newly introduced guards *APressure* and *SPressure* at both transitions, thus encoding their dynamic binding.
- **Statechart Simulator** 4: This view controls the statechart simulator. The developer can fire events, e.g., *MatSensorDetected*, and inspect/set values of variables, e.g., *SensorMetal*). Due to variability encoding, new (guard) variables have been introduced 8 and may be changed at run time. In this way, one can toggle between features *APressure* and *SPressure*. Although the developer may select an invalid feature combination w.r.t. the feature model 7 (i.e., select both features), no invalid statechart behavior is possible due to both transition guards checking their mutual exclusion (see Sect. 3).

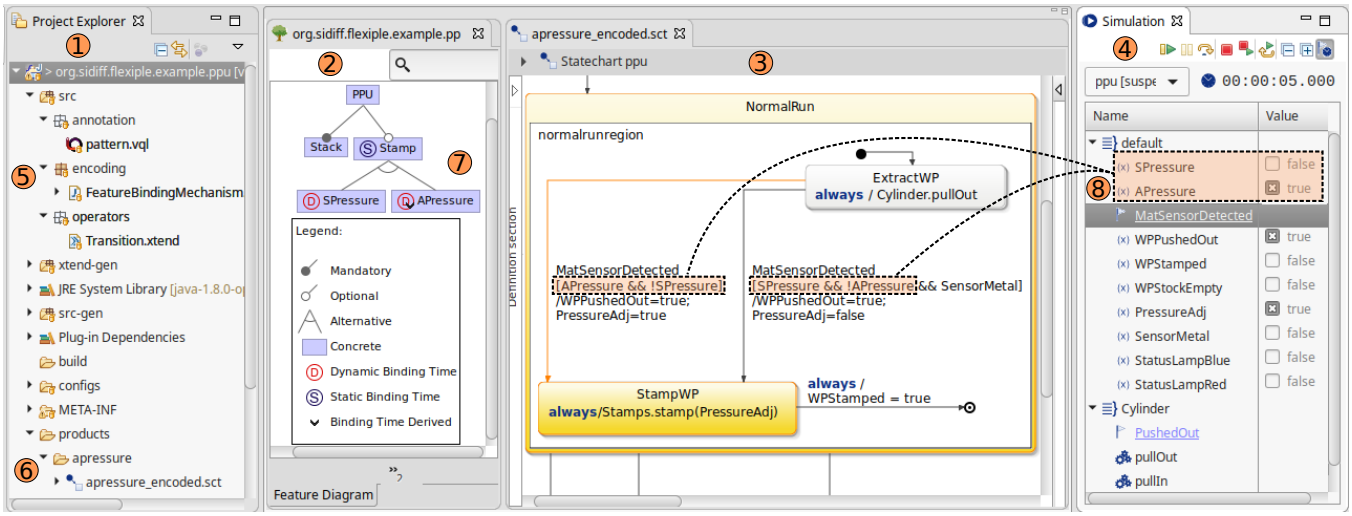


Figure 5: Tool Perspective of FlexiPLE. The resulting DSPL of the PPU allows for dynamic switching of features *SPressure* and *APressure* at run time (i.e., during simulation) due to encoding variability into (mutually exclusive) transition guards.

5 LIMITATIONS AND WORK IN PROGRESS

We are currently working on various limitations and enhancements of our tool. First, we are working on the support of complex presence conditions during variability analysis. Currently, FlexiPLE only supports single (possibly negated) feature annotations, which is a limitation in complex MBSPLs. Second, FlexiPLE does not take the current run time state of the model into account. Thus the developer may reconfigure/adapt the product configuration such that no viable paths in the current run state are available (anymore). Based upon our experiences made in earlier work and other contexts [9, 10], we finally plan to integrate other (executable) modeling languages, i.e., graph transformation languages.

6 RELATED WORK

Our approach is similar to [11, 12] which supports feature-wise dynamic binding in source-code-based SPLs. Unlike [11], we instantiate these concepts for modeling languages and offer integrated tool support for (the implementation of) dynamic binding. We integrate domain constraints at *all* binding times, as required in industrial contexts [3]. To this end, we encode (complex) constraints w.r.t. the feature model into the resulting product for ensuring that only valid feature combinations can be selected dynamically. In contrast to [12], the reconfiguration process itself is out of scope of our tool. Our approach builds upon the concept of variability encoding used in the context of software analysis [9, 14]. In these cases, code is adapted for enabling load time binding using *if* statements for encoding variability. In the context model-based development, several domain-specific encoding operators have been presented (i.e., class diagrams [10] or architecture models [5, 8]). Due to tight integration of state-of-the-art tools in the context of MBSPLs (i.e., the model query language and engine from Viatra) through all pipeline steps, FlexiPLE is capable of integrating new modeling languages as needed by implementing the respective artifacts (i.e., annotation mechanism pattern and VEOs).

ACKNOWLEDGMENTS

This work was partially supported by the DFG (German Research Foundation) within the CoMoVa project (grant nr 330452222) and under the Priority Programme SPP 1593: Design For Future – Managed Software Evolution.

REFERENCES

[1] 2019. FlexiPLE. http://pi.informatik.uni-siegen.de/projects/variance/flexiple.
[2] S. Apel, D. Batory, C. Kästner, and G. Saake. 2013. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer Science & Business Media, Berlin Heidelberg.
[3] Johannes Bürdek, Sascha Lity, Malte Lochau, Markus Berens, Ursula Goltz, and Andy Schürr. 2013. Staged configuration of dynamic software product lines with complex binding time constraints. ACM Press, 1–8.
[4] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2004. Staged Configuration Using Feature Models. In *Software Product Lines*, Robert L. Nord (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 266–283.
[5] T. Degueule, J. Filho, O. Barais, M. Acher, J. Le Noir, S. Madelénat, G. Gailliard, G. Burlot, and O. Constant. 2015. Tooling Support for Variability and Architectural Patterns in Systems Engineering *(SPLC '15)*. ACM, New York, NY, USA, 361–364.
[6] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. 2008. Dynamic Software Product Lines. *Computer* 41, 4 (April 2008), 93–95.
[7] Sascha Lity, Johannes Bürdek, Malte Lochau, Markus Berens, Andy Schürr, and Ina Schäfer. 2015. Re-Engineering Automation Systems as Dynamic Software Product Lines. *MBEES '15 Dagstuhl-Workshop* (2015).
[8] Marie Ludwig, Nicolas Farcet, Jean-Philippe Babau, and Joël Champeau. 2011. *Integrating Design and Runtime Variability Support into a System ADL.* Springer Berlin Heidelberg, Berlin, Heidelberg, 270–281.
[9] Dennis Reuling, Udo Kelter, Johannes Bürdek, and Malte Lochau. 2019. Automated N-way Program Merging for Facilitating Family-based Analyses of Variant-rich Software (accepted). *ACM Trans. Softw. Eng. Methodol.* (2019). https://doi.org/10.1145/3313789
[10] Dennis Reuling, Malte Lochau, and Udo Kelter. 2019. From Imprecise N-Way Model Matching to Precise N-Way Model Merging (accepted). *Journal of Object Technology* 18, 2 (2019), 1–20. https://doi.org/doi:10.5381/jot.2019.18.2.a8
[11] Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. 2011. Flexible Feature Binding in Software Product Lines. *Automated Software Engg.* 18, 2 (June 2011), 163–197.
[12] Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. 2011. Tailoring Dynamic Software Product Lines. In *Proceedings of GPCE (GPCE '11)*. ACM, New York, NY, USA, 3–12.
[13] B. Vogel-Heuser, C. Legat, J. Folmer, and S. Feldmann. 2014. *Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit.* Technical Report TUM-AIS-TR-01-14-02. TU München.
[14] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. 2016. Variability encoding: From compile-time to load-time variability. *Journal of Logical and Algebraic Methods in Programming* (2016).

HADAS: Analysing Quality Attributes of Software Configurations

Daniel-Jesus Munoz
CAOSD, Dpt. LCC, Universidad de Málaga, Andalucía Tech, Spain
danimg@lcc.uma.es

Mónica Pinto
CAOSD, Dpt. LCC, Universidad de Málaga, Andalucía Tech, Spain
pinto@lcc.uma.es

Lidia Fuentes
CAOSD, Dpt. LCC, Universidad de Málaga, Andalucía Tech, Spain
lff@lcc.uma.es

ABSTRACT

Software Product Lines (SPLs) are highly configurable systems. Automatic analyses of SPLs rely on solvers to navigate complex dependencies among features and find legal solutions. Variability analysis tools are complex due to the diversity of products and domain-specific knowledge. On that, while there are experimental studies that analyse quality attributes, the knowledge is not easily accessible for developers, and its appliance is not trivial. Aiming to allow the industry to quality-explore SPL design spaces, we developed the HADAS assistant that: (1) models systems and collects quality attributes metrics in a cloud repository, and (2) reasons about it helping developers with quality attributes requirements.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; *Abstraction, modeling and modularity*; Model-driven software engineering; *Software performance*; • **Computing methodologies → Modeling methodologies**; • **Information systems →** *Cloud storage*; • **Theory of computation → Automated reasoning**.

KEYWORDS

numerical, variability, model, software product line, attribute, NFQA

ACM Reference Format:
Daniel-Jesus Munoz, Mónica Pinto, and Lidia Fuentes. 2019. HADAS: Analysing Quality Attributes of Software Configurations. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3307630.3342385

1 INTRODUCTION

While traditional software development deals with only one variant at a time, SPLs consider, from the beginning, many possible variants simultaneously. SPLs aim to increase the quality of developed systems through reuse. Then, we need to explicitly model and manage the variability of the reusable assets. These models are called *Variability Models*, and its assets are called *Features*. A classical variability model contains Boolean features (e.g., $Compression\_functionality = True$) [4]. A *Numerical Variability Model* (NVM) additionally contains numerical features (e.g., $Microsoft\_Windows\_version = 10$) [3].

To find legal products of an SPL, hereafter called valid configurations, we rely on a *solver*. A solver navigates complex dependencies among the features of a variability model producing valid configurations [17]. *Complete valid configurations* (CVCs) are the ones where each of its features has a defined value. *Non-Functional Quality Attributes* (NFQAs) are linked to CVCs (e.g., cost, run-time).

In a software architecture there are *concerning features* that strongly affect NFQAs —e.g., adding *Security* functionality affects run-time. Additionally, a software is part of a heterogeneous environment, as it can be developed in a certain programming language, and runs in different operating systems and hardware. As concerns and environment are recurrent, HADAS NVM modelled them for NFQAs reasoning. Without HADAS, NFQAs research outcomes are disorganised, time consuming, and unequally distributed among diverse developers [14]. HADAS help developers to reason adjusting their products (i.e., CVCs) to NFQAs requirements without domain-specific knowledge. HADAS NFQAs assistant comprises:

- NVMs reasoning based on feature selection and a solver.
- A cloud repository that directly links NFQAs and meta-data with CVCs.
- A cross-platform web-app [1] with interactive NFQA graphs.
- A micro-service for third-party applications interaction.
- A set of plugins that integrate HADAS into widely used *Integrated Development Environments* (IDEs).

The rest of the paper is organised as follows. Section 2 describes two roles of HADAS' users. Section 3 details the tool. Section 4 contains a wide evaluation with different case studies. Finally, in Section 5 the conclusions and future work are presented.

2 USAGE OVERVIEW

HADAS aims to: (1) collect NFQAs research outcomes, and (2) help developers with NFQAs requirements based on the previous data.

Researchers and/or practitioners produce experimental data for different NFQAs (e.g., energy consumption of different CVCs with a multimeter). As it is unscalable to individually measure every (known) possibility in order to apply NFQAs requirements (e.g., runtime under a second), a collaborative approach is to collect available NFQAs measurements in a cloud database, as the HADAS *NFQAs repository* (Figure 1). HADAS offers an input interface in which a CSV file with the proper data structure must be uploaded (left side of Figure 1). Once uploaded, the NVM and the repository are populated, even though new features could be created.

Software Developers constantly have NFQAs stakeholder's requirements. However, it is not easy, nor timely wise, to discover

[1] HADAS NFQAs assistant web-app: https://hadas.caosd.lcc.uma.es/

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342385

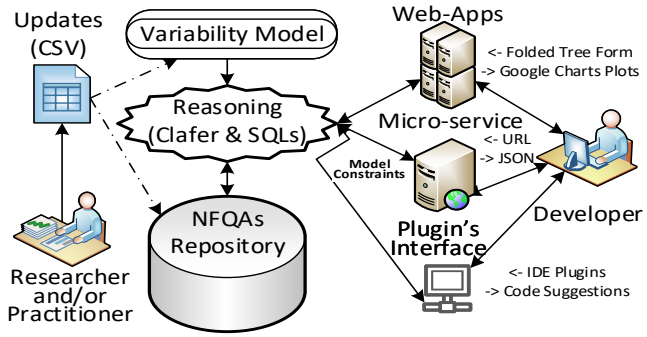


Figure 1: HADAS Overview and Usage

every alternative and/or potential improvement of their codes just by themselves [14]. As in the right side of Figure 1, they can use:

- **Web-view**: It is a user-friendly interface. The case study is selected in a tree view folded form that represents the HADAS NVM of Figure 2. After the reasoning process (i.e., solver and repository steps), the data is visualised as bar and line graphs per NFQA. The user is able to select which numerical feature is the X-Axis of the graphs, while the Y-Axis of the different graphs are the NFQAs. The default view is the complete NVM, but we are offering pre-reduced models under formal request using the main URL with subfolders (e.g., Waspmote view [2]).
- **Micro-service**: It is a low-level interface that allows external programs and services to directly interact with HADAS. A user selects the features through an URL. After reasoning, the micro-service returns the data in a JSON register with a nested structure similar to its NVM [11].
- **IDE plugins**: They are programmer-friendly interfaces. The most advanced is for JetBrains IDEs, due to their popularity [3]. While typing code, the plugin analyses source code searching for features whether they are explicit (e.g., Operations: Encrypt) or implicit (e.g., Operating System: Android). The search is based on the common programming libraries used in the current code (e.g., SpongyCastle for encryption). The programmer can select in the plugin which features are subject to change (i.e., adds constraints). Clicking the analysis button activates a reasoning request, creating an in-line comment within the code showing the fastest, the greenest and the most balanced configurations alongside their concrete features and meta-data [11] —meta-data helps to discard a conclusion (e.g., if the developer does not trust a measuring tool and/or practitioner, or if the data is considered too old).

As measurements of NFQA researchers grow, richer are the results.

3 TECHNICAL DETAILS

To model environment and concerns that affect NFQAs, HADAS uses the semantic layers of Figure 2. The HADAS NVM, repository, interactive form and CSV input data of Figure 1 share that structure:

[2]HADAS' Waspmote web-app view: https://hadas.caosd.lcc.uma.es/waspmote/

[3]https://www.g2crowd.com/categories/integrated-development-environment-ide

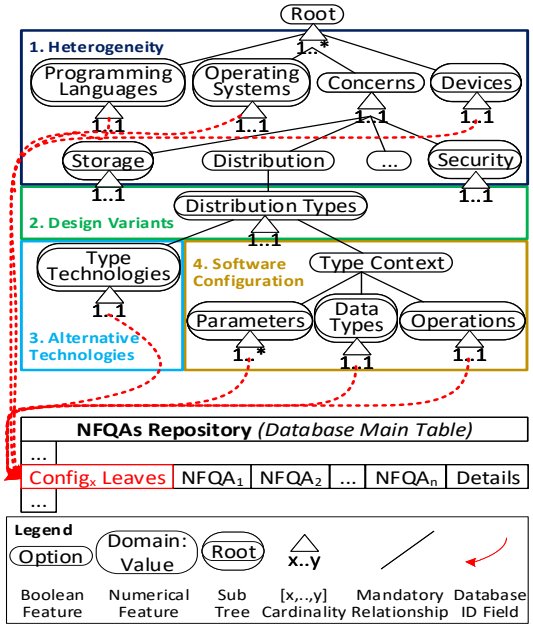


Figure 2: HADAS NVM and NFQAs Repository Structures

(1) **Heterogeneity**: Any application has purposes (i.e., concerns), is mainly developed in a programming language, and runs on a device and operating system. These distinctions decrease sub-tree repetitions [11].
(2) **Design Variants**: Any application can be differently designed depending on its concern (e.g., client/server vs P2P).
(3) **Alternative Technologies**: Any application can be developed using different technologies (e.g., libraries, algorithms).
(4) **Software Configuration**: Any application is configured to run with certain inputs, including default specifications.
  (a) **Parameters**: Any numerical feature is defined here (e.g., data size, number of requests, key length).
  (b) **Data Types**: E.g., Integer, String, List, Array.
  (c) **Operations**: Any application performs different operations (e.g., add, sort, store, decrypt).

To reason about the NVM we need an automated solver. There are three types of solvers: (1) *Constraint Programming* (CP), (2) *Satisfiability Modulo Theories* (SMT), and (3) *Satisfiability* (SAT) solvers. We discarded SAT solvers because currently they just support Boolean features. We have compared the two state-of-the-art solvers of the other two types, *Clafer* as CP solver, and *Z3py* as SMT solver. Clafer suite, besides other tools, contains a lightweight modelling language, and a model reasoner, which includes the CP library Chocosolver [1]. Z3py is a theorem prover which uses python modelling language and additionally supports bit-vectors, arrays, strings, and more [7]. To opt for the fastest solver, we have tested them randomly generating different number of samples for different already published models (ordered by size in Table 1), where the largest one is Busybox model with more than 10^{248} possible valid configurations. While both support our variability model structure, Clafer clearly outperforms Z3py in any model.

Table 1: Random Sampling Time Comparison in Seconds

	NVM	Z3py		Clafer	
Type	*# Samples:*	**300**	**500**	**300**	**500**
FSE2015 [16]	**Dune**	5.41	9.17	1.24	1.79
	HSMGP	14.30	23.60	1.51	2.30
	HiPAcc	5.97	9.92	1.35	1.93
	Trimesh	6.26	10.60	1.53	2.23
KConfig [2]	**axTLS**	34.63	58.30	2.38	3.01
	Fiasco	72.50	119.02	3.38	3.59
	uClibc-ng	81.10	135.01	6.33	7.67
	Busybox	341.39	576.10	188	202

Consequently, in HADAS, Figure 2 models are defined as Clafer models, and Clafer solver reasons over them. Numerical features and attributes are directly related to single features within the models [5]. In contrast, as NFQAs are related to CVCs [9], we need to link the model with a NFQAs repository as in Figure 1. NFQAs are modelled in the SQL (database programming language) repository, not in the Clafer NVM. Clafer solver alongside several back-end scripts can convert Clafer models into pairs of CVCs and SQL queries. HADAS has a limit of a 100 CVCs/SQL queries without considering numerical features variability in order to maintain concurrent performance and comprehensiveness of the results. The NFQAs repository is then able to execute those queries obtaining NFQA triples as `(name, value, details)` of each CVC. The details comprise everything not subject to vary in a configuration (e.g., measurement tool, date of measurement, extra information, meta-data), and pre-calculated statistics. As some statistics (e.g., correlations) are computationally expensive to calculate, and will slow down HADAS if they are executed in each user request, they are pre-calculated and added to the `details` field. HADAS repository is built upon a *MariaDB 10.2* relational database.

Currently, HADAS performs Pearson's chi-squared differentials and Bootstrapping statistics [12], to automatically check the significance of linear correlations of NFQAs with numerical features that can influence the selection of a particular configuration. It has two sequential steps: (1) calculate if a linear correlation exists between two features using the least squares regression method; (2) analyse if the established correlation is significant.

Technically, the repository identifies the triples with leaf features in the variability tree model representation of CVCs as in Figure 2. In other words, parent features are not stored in the repository exponentially decreasing the size of the database. This is possible due to the nature of the unique name of each feature within a model. Additionally, numerical features are fixed in the reasoning process and passed through as SQL constraints, allowing to retrieve the data in batches, improving HADAS performance by reducing the number of configurations to generate in the reasoning process and the number of database transactions.

Parameters (as in Figure 2) create the regular relational databases problem of *Entity-Attribute-Value* [4] (i.e., Name-Domain-Value), and has been solved with a partial repetition fusing two tables (i.e., Entity-(Attribute, Value)) decreasing the worst-case execution time from seconds [12] to milliseconds [11].

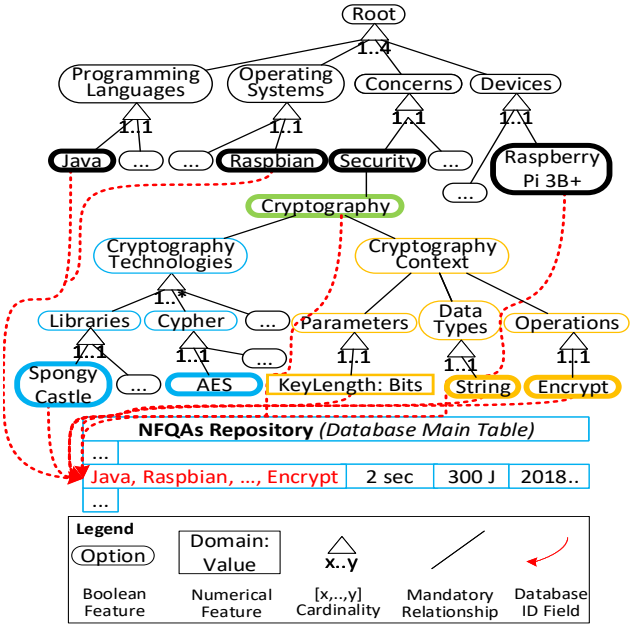


Figure 3: HADAS Security Case Study Example

Table 2: HADAS eco-assistant benchmarked

Configuration Size	# Configurations	Milliseconds
5	**10**	**4**
5	50	6
10	**10**	**10**
10	200	16
20	**10**	**32**
20	800	54

As previously explained, features and triples are differently selected and returned depending on the HADAS service. The interactive interface is a Twitter Bootstrap 3 [5] and Google Charts API [6] web-app. The micro-service access is through an URL containing nine arrays (one per model layer and context). In Figure 3 we have a reduced case study of a CVC of *Security* software for Raspberry Pis. The repository contains the leaf features of each of the nine layers selected (*Java, Raspbian, Raspberry Pi B3+,…*), as well as the time and energy consumption, the NFQAs for encryption (e.g., 2 seconds and 300 joules), and the date of measurement. If tested in HADAS, it will plot 4 Excel like graphs (line and bar graphs for time and for energy) where the *KeyLength* is the X-Axis. This allows to compare time and energy consumption in an increasing KeyLength for different cyphers (note the lack of AES link in Figure 3).

4 EVALUATION AND RESULTS

We have **benchmarked** HADAS with different testing NVMs in an Intel (R) Core i7-4790 CPU@3.60 GHz processor with 16 GB of

[4] Entity-Attribute-Value relational database issue: http://mysql.rjweb.org/doc.php/eav

[5] Twitter Bootstrap Framework 3: https://getbootstrap.com/docs/3.3/

[6] Google Charts API: https://developers.google.com/chart/

memory RAM and an SSD disk running an updated Ubuntu Server $18.04_{\times 64}$ LTS. As highlighted in Table 2, larger configurations mean larger run-times. Although, these results confirm that there is a bottleneck in the NFQAs repository, as a large number of foreign keys slows-down a database, the run-times are still within the range of milliseconds. Also, is unlikely that a developer wish/is able to compare graphs of complete colossal SPLs at once (e.g., $10^8 0$ CVCs). Network, web-explorers and plugins overheads are negligible.

We have evaluated our approach in terms of usability and usefulness [11]. A total of 17 participants with different ages and backgrounds participated in an evaluation test and a **survey** using a Liker-scale where the least positive answer is 1, with 5 being the most positive. A summary is that they had not considered a solution like HADAS before (0.4/5); the functionality seems user-friendly (3.6/5) and the information is easy to understand (3.9/5). The interface is consistent (3.9/5) and the variability is clear (3.9/5). They would recommend it and make use of it (3.8/5).

As focus on green computing has been growing rapidly among developers in the past few years [18], it is the perfect niche to test HADAS. *Green Computing* refers to use computing resources more efficiently without negatively affecting the overall performance. The most important NFQAs in this context are run-time and energy consumption [10]. We have based our case studies in works of embedded devices [15], client/server distribution [6], programming language frameworks [8], etc.

We have applied our approach to two different case-studies. In the first one [12], the most common **web servers** and their intrinsic relationship with energy consumption and performance is analysed. Where Nginx is always the greenest alternative, and Apache the fastest, if a trade-off is necessary, Nginx running on Linux and ISS running on Windows are the most balanced solutions. Additionally, there is a significant linear relationship of energy and time with the number of users and page size, but much stronger in the case of the number of users. In the seconds case **cyber-physical systems** were analysed [11]. We discovered that we could reach energy savings up to 70% if sending data in the largest possible batches through Wifi. However there is a cross between Wifi and Bluetooth CVCs. For small and mid-size batches, Wifi is the most energy-efficient option, leaving Bluetooth for large batches. More and richer results can be consulted in the citations.

5 CONCLUSIONS AND FUTURE WORK

We have presented the concept of NVMs and NFQAs assistant, mainly tested in green computing research and development. HADAS aims to collect NFQAs research outcomes, and help developers with NFQAs requirements to reason about their best alternatives, even considering the hardware. The main artefacts are: (1) the HADAS NVM for SPLs structure, (2) the NFQAs cloud repository, (3) the connection of NVMs and NFQAs with a solver using Clafer modelling language and SQL queries, and (4) the globally accessible web-services. HADAS has been benchmarked and tested by professional developers, assessing its scalability, user-friendliness and usability. HADAS analyses could lead to an energy efficiency improvement of a 70% in cases like cyber-physical systems, or, in a lower degree, web servers. As future work this tool can: (1) be easily extended to automatically support other NFQAs, (2) implement a larger and more advanced correlation statistics, and (3) introduce estimation techniques as random sampling to reduce the need of physical measurements to reason over the models and NFQAs [13].

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).

REFERENCES

[1] MichałAntkiewicz, Kacper Bąk, Alexandr 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

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Science & Business Media, Berlin, Heidelberg.

[3] Muhammad Ali Babar, Lianping Chen, and Forrest Shull. 2010. Managing variability in software product lines. *IEEE software* 27, 3 (2010), 89–91.

[4] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Software Product Lines*, Henk Obbink and Klaus Pohl (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 7–20.

[5] David Benavides, Sergio Segura, and Antonio RuizCortés. 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

[6] S. A. Chowdhury, V. Sapra, and A. Hindle. 2016. Client-Side Energy Efficiency of HTTP/2 for Web and Mobile App Developers. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, Vol. 1. IEEE Computer Society, Washington, DC, USA, 529–540. https://doi.org/10.1109/SANER.2016.77

[7] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, Berlin, Heidelberg, 337–340.

[8] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. Energy Profiles of Java Collections Classes. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 225–236. https://doi.org/10.1145/2884781.2884869

[9] JoseMiguel Horcas, Mónica Pinto, and Lidia Fuentes. 2018. Variability models for generating efficient configurations of functional quality attributes. *IST* 95 (2018), 147–164. https://doi.org/10.1016/j.infsof.2017.10.018

[10] V. Kharchenko, A. Gorbenko, V. Sklyar, and C. Phillips. 2013. Green computing and communications in critical application domains: Challenges and solutions. In *The International Conference on Digital Technologies 2013*. IEEE Computer Society, Washington, DC, USA, 191–197. https://doi.org/10.1109/DT.2013.6566310

[11] Daniel-Jesus Munoz, 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

[12] 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 (01 Nov 2018), 1155–1173. https://doi.org/10.1007/s00607-018-0632-7

[13] 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

[14] C. Pang, A. Hindle, B. Adams, and A. E. Hassan. 2016. What Do Programmers Know about Software Energy Consumption? *IEEE Software* 33, 3 (May 2016), 83–89. https://doi.org/10.1109/MS.2015.83

[15] C. Shen and M. Srivastava. 2017. Hardware Heterogeneity to Improve Pervasive Inferences. *Computer* 50, 6 (2017), 19–26. https://doi.org/10.1109/MC.2017.174

[16] 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

[17] Thomas Thum, Don Batory, and Christian Kastner. 2009. Reasoning About Edits to Feature Models. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 254–264. https://doi.org/10.1109/ICSE.2009.5070526

[18] D. Wang. 2008. Meeting Green Computing Challenges. In *2008 10th Electronics Packaging Technology Conference*. IEEE Computer Society, Washington, DC, USA, 121–126. https://doi.org/10.1109/EPTC.2008.4763421

Change Analysis of #if-def Blocks with *FeatureCloud*

Oscar Díaz
University of the Basque Country
San Sebastián, Spain
oscar.diaz@ehu.eus

Raul Medeiros
University of the Basque Country
San Sebastián, Spain
raul.medeiros@ehu.eus

Leticia Montalvillo
Ikerlan Research Center
Arrasate, Spain
lmontalvillo@ikerlan.es

ABSTRACT

FeatureCloud is a git client for the visualization of evolution in annotated SPL, i.e. those resorting to ifdefs for variability. Specifically, FeatureCloud (1) mines git repositories; (2) extracts ifdefs; (3) works out differences between two versions of the same ifdefs; (4) abstracts ifdefs in terms of their code churn, tangling and scattering; and finally (5), aggregates and visualizes these properties through "feature clouds". Feature clouds aim to play the same role for SPLs that "word clouds" for textual content: provide an abstract view of the occurrence of features along evolving code, where "repetition" account for scattering and tangling of features. Here, we introduce the analysis goals, the perspective (i.e. the object of analysis) and visualization strategies that underpin FeatureCloud.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software evolution**;

KEYWORDS

SPL Evolution, Preprocessor Directives, Tag Clouds

ACM Reference Format:
Oscar Díaz, Raul Medeiros, and Leticia Montalvillo. 2019. Change Analysis of #if-def Blocks with *FeatureCloud*. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3307630.3342386

1 INTRODUCTION

Annotated SPLs resort to preprocessor directives (a.k.a. #if-def blocks or just *"ifdefs"*) to realize variability in code. This is a powerful approach to readily support variability but at the expenses of complicating evolution and understandability [4]. This problem exacerbates if the interest is not on a snapshot of the code but on how this code evolves, i.e. change analysis. Here, we focus on two main questions about SPL evolution: (1) which are the hot-spot features, i.e. features with the most complex upgrades during the last release; and (2), how feature scattering and tangling are evolving.

We address these question through ***FeatureCloud***, a Web client that mines and visualizes *Git* repositories containing *ifdefs*. *FeatureCloud* is based on two main insights:

- *ifdefs* are code but *annotated* code, i.e. code governed by the presence of feature constants in their preprocessor directive. Web 2.0 made popular to *annotate* resources (e.g. pictures) with tags. If we consider code blocks as resources, then *ifdefs* or more specifically, feature constants might play the role of tags that describe when the code block is to be included. On these premises, we can tap into familiar tag visualization approaches, e.g., tag clouds [8].
- maintenance wise, code churn rather than the lines of codes (LOC), are suggested as a better indicator of the effort of developers [5, 9]. Code churn is a metric measuring change volume between two versions of a system, defined as the sum of added and deleted lines [6]. Likewise, we introduce the notion of "#ifdefs churn" (hereafter denoted as $\triangle ifdefs$) as the LOC added/deleted in the context of a given #ifdef block.

The rest of the paper characterizes *FeatureCloud* along the Software Visualization Framework proposed in [7]*:* analysis goals, perspective and visualization strategy. *FeatureCloud* is Open Source. Drivers are provided for *pure::variants* and C annotations. A video is available at https://vimeo.com/338522586.

2 GOALS

This refers to the analysis rationales that underpin the visualization effort. *FeatureCloud* considers two main analysis goals:

- identifying hot-spot features , i.e. features with the most complex upgrades. This might help to improve the efficiency of the testing process by driving the attention to these hot-spots. Human resources wise, hot-spot identification might serve to determine which of the organization's feature units (i.e. domain engineers) or product units (i.e. application engineers) have been impacted to a larger extent by the SPL's evolution
- identifying dissonance between the problem space (i.e. the feature model) and the solution space (i.e. feature realization through #ifdefs). Development urgency during SPL evolution might lead new feature dependencies to be reflected uniquely in the code without being mirrored in the feature model.

Hence, we do not consider the Feature Model but extract features and infer dependencies out of the #ifdef blocks in which the feature constants are embedded.

3 PERSPECTIVES

A perspective represents "a set of coordinated views designed to represent a group of properties of the object of analysis" [7]. Whereas the literature mainly considers #ifdef blocks as the object of analysis [3], our focus is not on the SPL's current state but on how the SPL evolves. Accordingly, we tackle no #ifdef snapshots but #ifdef

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342386

Figure 1: A $\triangle ifdef$ stands for a DIFF between two versions of the same #ifdef block. In the figure, the $\triangle ifdef$ corresponds to the LOCs with green/red background.

deltas ($\triangle ifdefs$), i.e. differences between two versions of the same #ifdef block. Deltas are worked using traditional DIFF utilities (see Fig. 1). Next, this object of analysis needs to be characterized along a set of metrics. To this end, we resort to the same metrics used for #ifdef blocks (in brackets primary studies referring to the metric in El-Sharkawy et al.'s survey):

- tangling, i.e. how many feature constants are involved in $\triangle ifdef$ directives (6 primary studies). For the $\triangle ifdefs$ in Fig. 1, its tangling is 3[1]
- scattering, i.e. how many $\triangle ifdef$ are affected by a given feature constant (10 primary studies),
- lines of code of $\triangle ifdefs$. This reflects the change volume between two $\triangle ifdef$ versions, defined as the sum of added and deleted LOC (0 primary studies[2]). For the $\triangle ifdefs$ in Fig. 1, its LOC is 10.

A lot of work goes into working out $\triangle ifdefs$ out of #ifdefs. We need first to work out the set of code churn for two SPL releases. Next, obtain the metrics for the set of code churn. And finally, visualize those properties. The latter should not be understated. It is the visualization paradigm the one that will end up supporting the analysis workflow, and its suitability will determine the final adoption of the tool. Two main components of the visualization approach are the visual paradigm and the mechanisms of interaction.

3.1 The visual paradigm

FeatureCloud conceptualizes $\triangle ifdef$ as resources, and feature constants as tags that describe these resources. On these premises, we apply tag-cloud theory to the visualization of the properties of *#ifdef churn* [1]. Specifically, we propose **"feature clouds"**. Similar to word clouds, feature clouds organize features along a tree of nodes (see Fig. 2). Node rendering, i.e. their size, arrangement and color, are used to account for $\triangle ifdefs$ metrics, namely:

- node arrangement reflects the tangling, i.e. feature constants are arranged on a tree to reflect "their tangling proximity",
- node size stands for the scattering as an attempt to reflect the number of #ifdefs where the feature appears,
- node color accounts for the lines of code, i.e. color fading from light blue (small LOC) to dark blue (large LOC).

As an example, consider the 3D-printer firmware Marlin[3]. Marlin relies on the C preprocessor to support #ifdefs. Fig. 2 shows the feature cloud for the changes conducted from Release 1.1.7 to Release 1.1.8 as mined from the Marlin's *git* repository. Some comments are due.

First, node arrangement. This facet tackles eventual tangling concerns by highlighting "clusters of features" that tend to appear together. The strength of this tendency is reflected by the proximity of the feature nodes. If nodes branch off the same point (Fig 2 (a)), then features always appear together in the $\triangle ifdefs$ at hand (i.e. tight tangling). If nodes appear in the same branch but from different spurs (Fig 2 (b)), then feature constants show up together in some $\triangle ifdefs$ but not in others (i.e. medium tangling). Finally, if a node is displayed alone in a branch (Fig 2 (c)), then the corresponding feature constant always appear alone in the $\triangle ifdefs$ being considered.

Second, node size. No matter whether a feature constant appears alone or tangled, it can be scattered along different files. Node size reflects this metric. Hence, feature NOZZLE_PARK_FEATURE is by far the one most scattered.

Third, node color. Color opacity stands for the LOCs being changed (i.e. code churn). This provides clues about potential hotspots, i.e. features with high activity. Though LOC might not impact understandability to the same extent as the other measures, it is an indicator of programming effort, and hence, resources being dedicated.

3.2 The mechanisms of interaction

Visualization does not only refer to the graphics. The interaction scaffold that facilitates the analysis workflow is paramount to ease adoption. *FeatureCloud* supports a feature-based workflow that drills down from feature constants to the raw $\triangle ifdefs$ code. This is realized through three Graphical User Interfaces (GUIs). Analysts can go back and forth between the distinct GUIs to explore different branches of the feature cloud.

3.2.1 The Tangling GUI . It is the starting point (see Fig. 2). It provides an overview of the upgraded features as well as potential "change clusters" that can be derived from the feature cloud. The opacity of the node color provides a glimpse of the effort involved in terms of code churn. By clicking on a feature node, analysts move to the Scattering GUI.

At this state, main concern is the analysis scope. SPLs not only evolve in time but also in space [2]. Existing *#ifdefs* might need to be enhanced to cope with products' changing requirements (i.e., evolution in time). In addition, new features might need to be added as the SPL's scope expands (i.e., evolution in space). This introduces two main concerns during change analysis: time window (i.e., evolution in time) and feature scope (i.e., evolution in space). Both parameters are set from this GUI.

Evolution in time. This refers to the time window for which $\triangle ifdefs$ are calculated. $\triangle ifdefs$ are worked out from two *git* commits that set the temporal scope of the change to be analyzed . Fig.

[1]Working out tangling is further complicated by the presence of nested #ifdef blocks. *FeatureCloud* considers nesting.
[2]El-Sharkawy et al. survey reports 10 primary studies that consider LOCs but none considers the notion of code churn.
[3]https://github.com/MarlinFirmware/Marlin

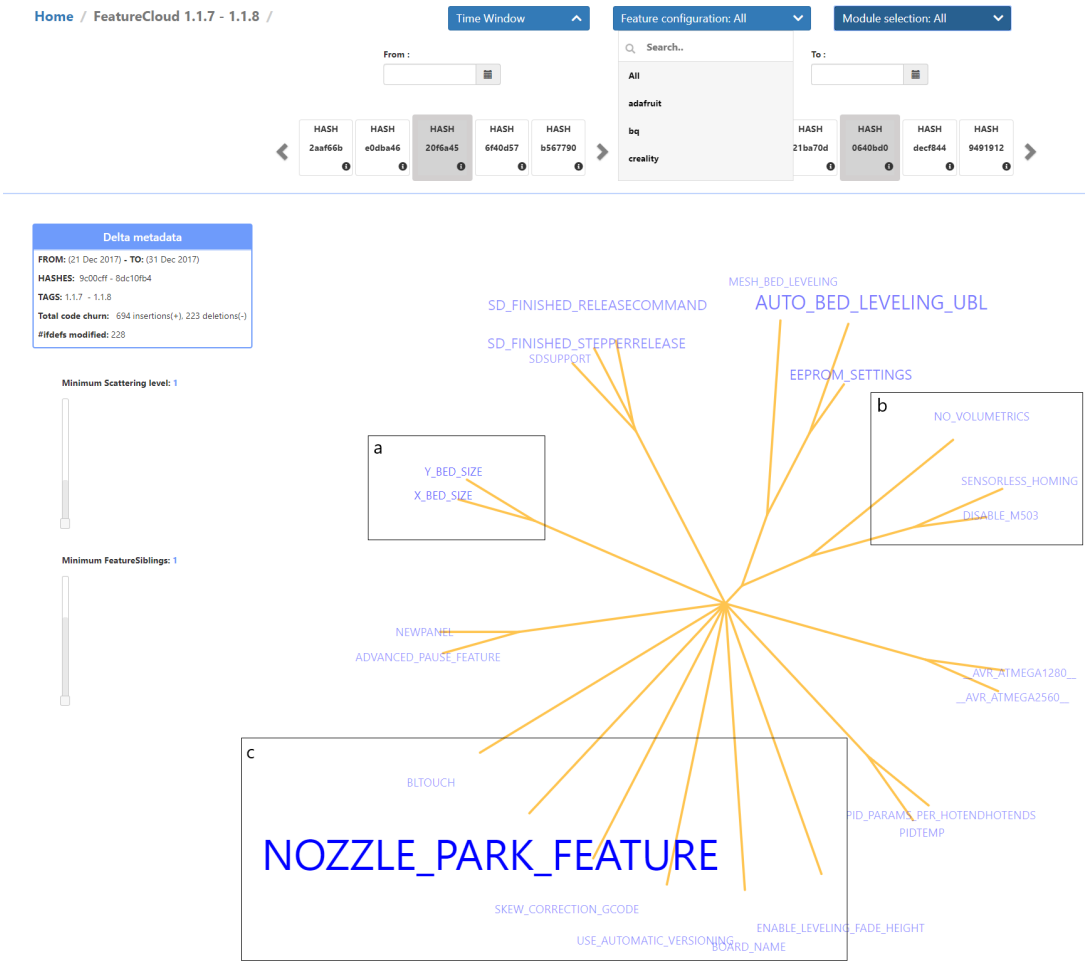


Figure 2: The Tangling GUI for Marlin's timespan "Release 1.1.7-to-Release 1.1.8". Click on a feature node (e.g. EEPROM_SETTINGS) to move to the Scattering GUI for this feature.

2 displays the calendar widget use for this purpose. Analysts select both the "from" commit and the "to" commit for which the churn is to be worked out.

Evolution in space. The scope of the change can be set in terms of the features to be tracked. Domain Engineers are not always concerned about all features but just those feature clusters they are responsible for. On the other hand, Application Engineers might focus their interest uniquely on their products' features. Accordingly, *FeatureCloud* supports configuration-based analysis. That is, analysts limit their interests to those features confined to a given configuration. Hence, Application Engineers provide their product configuration as input. Likewise, Domain Engineers need to create "spurious configurations", i.e. configurations whose features are not those of any product but collect the features the Domain Engineer is interested in tracking.

3.2.2 The Scattering GUI . This interface shows the different preprocessor directives in which a feature constant shows up (see Fig. 3). A Feature Sibling Group (FSG) collects those groups of features that appear together in at least one preprocessor directive for the $\triangle ifdefs$ being analyzed. FSGs are similar to the notion of Variation Point Group (VPG) in [10]. A VPG is a group of VPs with *equivalent predicate* of selecting variant code fragments. Unlike FSGs, a VPG retains the boolean expression that tights the features together whereas FSGs only consider the presence of the feature constant, no matter the boolean expression. The rationales are that boolean expressions might provide a too fine-grained detail that might not be needed when the aim is to get a glimpse of the extent of how much a feature have been upgraded.

Analysts might ponder a feature's "maintainability proneness" by looking at the number of places and companion features this feature is involved. Fig. 3 shows the case of EEPROM_SETTINGS. This GUI helps to assess maintainability risks in terms of *the spread* of the change. Next, analysts can delve into the change itself, i.e. the LOC.

3.2.3 The Code GUI . FSG introduces an equivalence class among $\triangle ifdefs$: those $\triangle ifdefs$ with the same feature constants in their

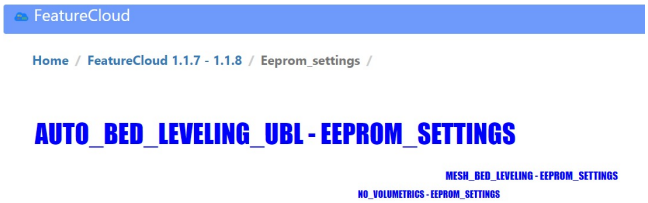


Figure 3: The Scattering GUI for EEPROM_SETTINGS. This feature is being combined with three other features. Node size reflects changed LOC . Click on an FSG node to move to the Code GUI for this FSG.

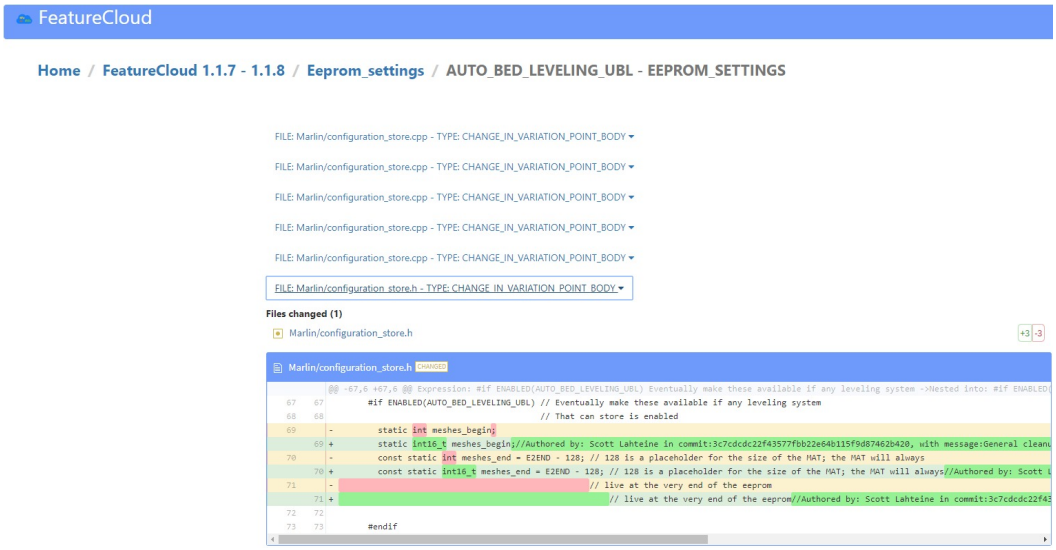


Figure 4: The Code GUI for the AUTO_BED_LEVELING_UBL - EEPROM_SETTINGS. Changes are scattered along six $\triangle ifdefs$.

directives. The Code GUI displays this equivalence class for the selected FSG (see Fig. 4). $\triangle ifdefs$ are displayed using a DIFF notation: added/deleted LOCs.

4 CONCLUSIONS

We introduced "feature clouds", a representation of change in #ifdef blocks. A namesake tool realizes this vision for *C*-preprocessor directives, and *pure::variants* as the annotation technology. *FeatureCloud* takes upgrades in #ifdef blocks (i.e. $\triangle ifdefs$) as the object of analysis. These objects are characterized along three main metrics: feature tangling, feature scattering and LOC. Analysis workflow follows a drill-down approach from feature constants down to the raw $\triangle ifdefs$ code. Formative evaluation is being conducted for different Open Source variability-intensive applications.

Acknowledgement. This project has benefited from a grant from *pure::systems'* GoSPLC initiative. This work is co-supported by MCIU/AEI/FEDER,UE under contract RTI2018-099818-B-I00 and MCIU-AEI TIN2017-90644-REDT.

REFERENCES

[1] Scott Bateman, Carl Gutwin, and Miguel Nacenta. Seeing things in the clouds. In *Proceedings of the nineteenth ACM conference on Hypertext and hypermedia - HT '08*, page 193, New York, New York, USA, 2008. ACM Press.

[2] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product derivation in software product families: A case study. *Journal of Systems and Software*, 74(2 SPEC. ISS.):173–194, 2005.

[3] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information and Software Technology*, 106:1 – 30, 2019.

[4] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. Do #ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 65–73, New York, NY, USA, 2016. ACM.

[5] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Software Eng.*, 38(6):1276–1304, 2012.

[6] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 284–292, New York, NY, USA, 2005. ACM.

[7] Renato Lima Novais, André Torres, Thiago Souto Mendes, Manoel Mendonça, and Nico Zazworka. Software evolution visualization: A systematic mapping study. *Information and Software Technology*, 55(11):1860–1883, nov 2013.

[8] James Sinclair and Michael Cardew-Hall. The folksonomy tag cloud: when is it useful? *J. Information Science*, 34(1):15–29, 2008.

[9] Dag I. K. Sjøberg, Aiko Fallas Yamashita, Bente Cecilie Dahlum Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Trans. Software Eng.*, 39(8):1144–1156, 2013.

[10] Bo Zhang, Martin Becker, Thomas Patzke, Krzysztof Sierszecki, and Juha Erik Savolainen. Variability evolution and erosion in industrial product lines. In *Proceedings of the 17th International Software Product Line Conference on - SPLC '13*, page 168, New York, New York, USA, 2013. ACM Press.

Understanding Conditional Compilation Through Integrated Representation of Variability and Source Code

David Baum
Christina Sixtus
Lisa Vogelsberg
Ulrich Eisenecker
david.baum@uni-leipzig.de
wir13dvx@studserv.uni-leipzig.de
ges11eso@studserv.uni-leipzig.de
eisenecker@wifa.uni-leipzig.de
Leipzig University
Leipzig, Germany

ABSTRACT

The C preprocessor (CPP) is a standard tool for introducing variability into source programs and is often applied either implicitly or explicitly for implementing a Software Product Line (SPL). Despite its practical relevance, CPP has many drawbacks. Because of that it is very difficult to understand the variability implemented using CPP. To facilitate this task we provide an innovative analytics tool which bridges the gap between feature models as more abstract representations of variability and its concrete implementation with the means of CPP. It allows to interactively explore the entities of a source program with respect to the variability realized by conditional compilation. Thus, it simplifies tracing and understanding the effect of enabling or disabling feature flags.

CCS CONCEPTS

• **Human-centered computing** → **Visual analytics**; *Information visualization*; • **Software and its engineering** → *Maintaining software.*

KEYWORDS

conditional compilation, variablity, software visualization, visual analytics, Getaviz, preprocessor, software prodect line

ACM Reference Format:
David Baum, Christina Sixtus, Lisa Vogelsberg, and Ulrich Eisenecker. 2019. Understanding Conditional Compilation Through Integrated Representation of Variability and Source Code. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3307630.3342387

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342387

1 INTRODUCTION

Conditional compilation is a way of introducing variability to C source code immediately before compile time. The CPP can be used to include or exclude source code components, which change the structure and behavior of the resulting program. Often Boolean feature flags are used to design complete SPLs. The complexity created by the numerous variants is challenging. Although feature models help to describe the variability, they are of limited use when working with the source code directly, e.g., during bug fixing. In general, bugs that lead to unwanted runtime behavior are often more difficult to detect and to fix than compile time errors. This applies even more if a bug only occurs under certain feature configurations. For this reason, the developer needs support for answering the following questions, that appear regularly during development: **Q1:** What effect does the activation of a feature have on the structure of a program? **Q2:** Which elements are contained in the source code given a certain feature configuration? With these questions in mind we developed an interactive analytics tool that provides the following functionality:

(1) It provides an overview over the structure of the system, i.e., all functions, global variables, and complex types that can be part of any variant.
(2) The user can define a set of flags and explore the structure of the resulting variant. This includes method calls, read and write operations, as well as the original C code.
(3) The analysis runs fully automated without any manual preparation steps.

A demo is available online[1]. Additionally, the usage of the tool is demonstrated in a screencast[2]. We first examine how existing tools support the presented use case, followed by a presentation of our tool. We will address several design choices, including variability extraction, and the visualizations the user interface is based on. A small application scenario based on the online demo is presented in chapter 5. Finally, we will discuss our previous experiences with the tool and future development.

[1] http://softvis.wifa.uni-leipzig.de/splc2019
[2] http://softvis.wifa.uni-leipzig.de/splc2019screencast

2 RELATED WORK

Most work on SPLs and variability either focuses on automatic checks at compile time or provides abstract models without a direct connection to the source code. In the area of C code refactoring numerous works can be found, that take preprocessor statements into account [4, 7, 15, 22, 25]. Feature models are often used to prepare the extracted information. Badros and Notkin have written a tool that analyzes unpreprocessed C source code with simple scripts [1]. The SPL community offers a number of tools for visualization and for better understanding variability points and variants. For example, two Eclipse plugins visualize feature models and perform type checking of preprocessor code [17, 23]. With the help of Meta Programming System (MPS) different views for editing and understanding SPL source code can be provided to developers [5]. Other tools generally support the development of SPLs without the need for specific focus on C source code. Feigenspan et al. have developed an Eclipse plugin that enables highlighting of feature code [6]. Nestor et al. have created visualizations for the configuration of SPLs [20], but they do not provide a direct connection to the source code. The Feature Relation Graph presents possible feature combinations depending on a selected feature [16]. Illescas et al. as well as Urli et al. show different visualization models for feature combinations but without a connection to the source code [9, 24]. Many works are based on the same extraction tools such as FeatureCoPP [12], SuperC [8], TypeChef [10], and Yacfe [21]. We are not aware of any tool that supports the presented use case satisfactorily. In the area of SPLs, the focus of research is on the representation of variability points and legal combinations of features. In most cases links to the underlying source code are not presented.

In contrast, some tools are aimed at improving the developer's understanding of the code. Livadas and Small have created an integrated development environment (IDE) extension that can be accessed by clicking a macro expansion. It shows where a macro has been defined and how the macro is expanded [14]. Also Kullbach and Riediger visualize macro expansion and conditional compilation with an IDE extension by so-called folding. When clicking on a preprocessor instruction, the corresponding precompiled source code is collapsed [13]. However, these tools are only useful for local contexts and do not address systemwide variability.

3 VARIABLITY EXTRACTION

Comprehensive preprocessing of the C code and the CPP statements is required to provide answers to the questions that have been raised. Our requirements on such a parser can be summarized as follows:

(1) The result of the parsing must contain all the linguistic means of the C standard. This includes translation units, functions, elementary types, complex types, information about function calls as well as reading and writing of global variables.
(2) The parser should consider the included files to handle declarations correctly.
(3) Macro expansions should be performed before parsing since the content of the macros may influence feature detection and location.
(4) The result of the parsing should contain information about the conditional compilation, including the CPP directives extracted from the source code. Even more useful would be an evaluation of nested conditions and an explicit representation of alternatives as distinct branches in the result.

There exist various tools with different scopes to analyze code with conditional compilation. We came to the conclusion that TypeChef meets our requirements best, although it is significantly slower than, e.g., SuperC. The goal of the developers of TypeChef was to create a complete and solid parser that can parse C code without manual preprocessing. It uses an LL parser to create an abstract syntax tree (AST) which contains all of the variability information we need. We modified TypeChef to serialize the complete AST to an XML file for further processing with jQAssistant. This is a program for analyzing and visualisizing software artifacts [18]. It is built on top of Neo4j, a graph database. We implemented a plugin for TypeChef to include C code and feature flags. The result is a graph containing all code entities, method calls, read and write accesses, features, and their dependencies.

4 USER INTERFACE

Getaviz [3] is an open source toolkit for visual software analytics [3]. It uses jQAssistant as information source and supports the automatic generation of visualizations for different use cases [2]. Getaviz comes with a highly configurable browser-based user interface for viewing and interacting with a visualization. Getaviz can be easily expanded to support new visualization types and interaction components. Hence, we used Getaviz as starting point and customized it to fit our requirements.

Figure 1 shows the default view containing a visualization of the structure (I), a search bar (II), the *FeatureExplorer* (III), and the *CodeViewer* (IV). To understand the structure and the included variablity it is useful to get an overview of the complete system first. Therefore, we visualize the structure in such a way that it can be fully grasped at a glance. This view contains all code entities that could be potentially compiled. Our prototype is based on the Recursive Disk (RD) metaphor [19]. It is designed to visualize the structure of imperative programming languages, with an emphasis on object-oriented languages, especially Java. As the name indicates, an RD visualization consists of nested disks, where each disk represents a package or a class in Java. In order to apply the visualization to C code, we had to make several changes. We chose translation units as top level elements replacing packages. They are depicted as gray disks as shown in Figure 2. A translation unit can contain multiple structs, unions, enums, global variables, and functions. Functions are depicted as blue segments. The area of a blue segment is proportional to the lines of code of the corresponding function. Variables are depicted as yellow segments that have a fixed size. Structs, enums, and unions are depicted as purple disks. They can contain further elements according to the content of the C entities. We have retained the original layout algorithm. All disks are ordered by size and then placed spiral-shaped clockwise around the largest disk. Although at first glance it seems chaotic, the emerging visual patterns and empty spaces give each disk a unique appearance and help the user to recognize specific disks.

The visualization is interactive, so the developer can easily explore it. The *FeatureExplorer* contains all extracted feature flags of the system. The developer can select or deselect individual flags

[3] https://github.com/softvis-research/Getaviz

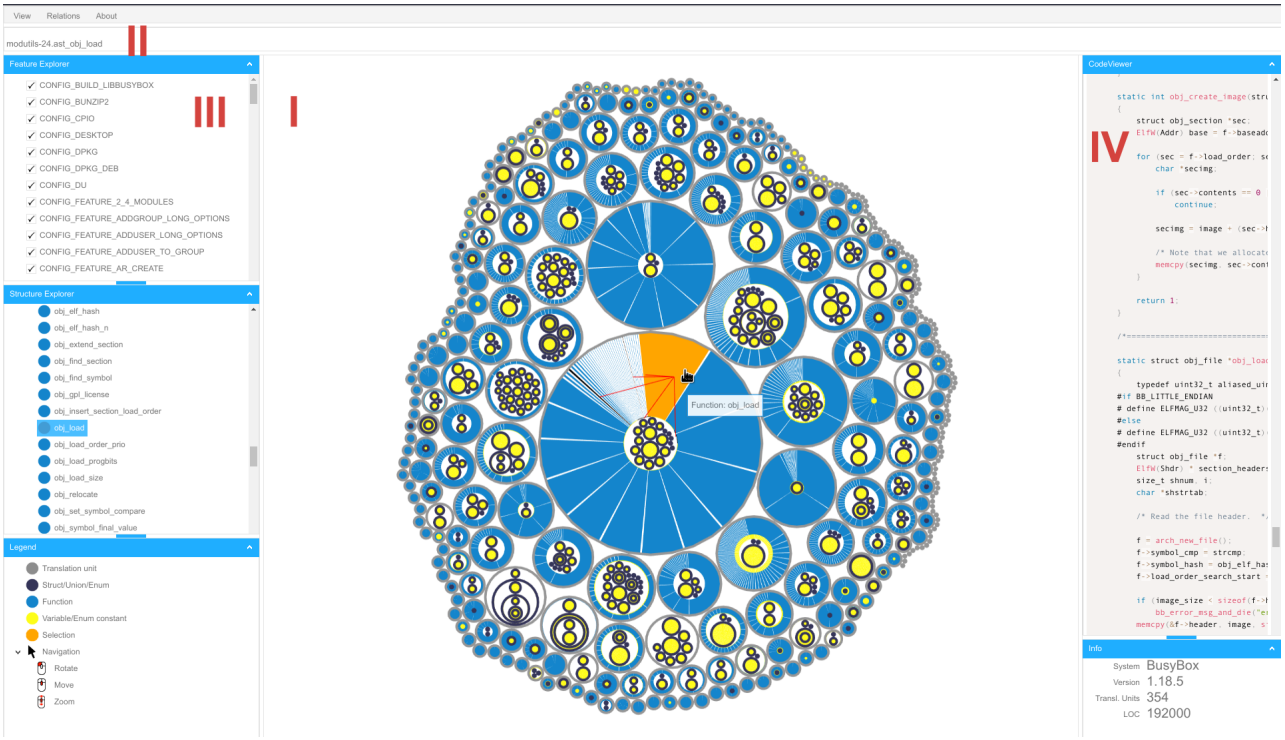


Figure 1: Screenshot of Getaviz visualisizing the structure of BusyBox

and the visualization gets updated accordingly. If the code entity is to be excluded by the CPP, then the graphical representation will be displayed transparently. In this way, the user can explore and understand the impact of the different flags to answer Q2 without having to jump from source file to source file and manually evaluate macros.

Detail information is provided as tooltip. In Figure 1, the method `obj_load` is selected and therefore highlighted orange. The red lines represent method calls and variable accesses of this method. Nevertheless, the source code is still of great interest for the developer since it is the main artifact to work with. To provide more context it is possible to view the source code directly in Getaviz. The *CodeViewer* on the right side displays the source code of the selected entity.

5 APPLICATION

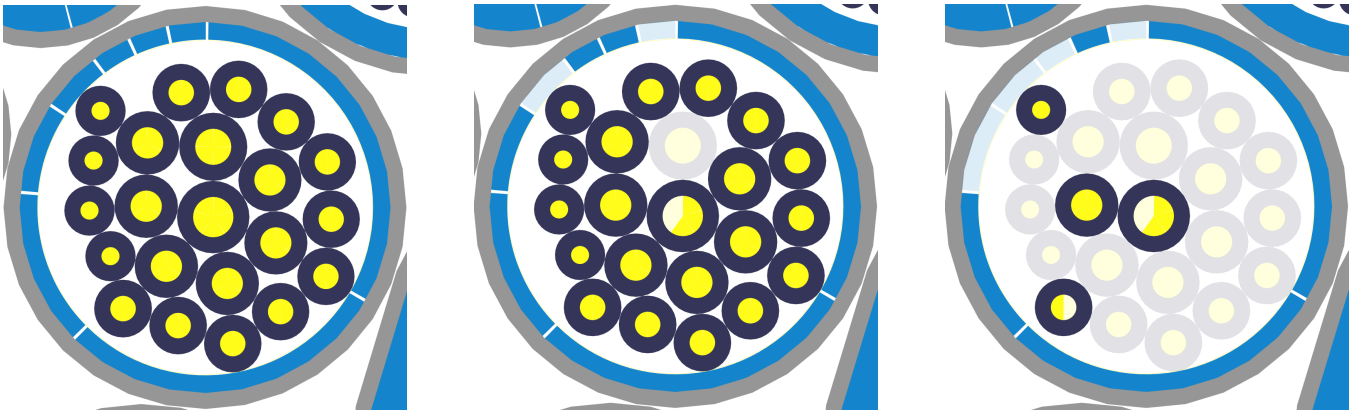
To demonstrate the usefulness of our tool we chose BusyBox 1.18.5 since it is a highly customizable system. It contains several hundreds of explicitly declared Boolean compile-time configuration options with complex dependencies [11]. One of these feature flags is `CONFIG_DESKTOP` which affects the macros `ENABLE_DESKTOP`, `IF_DESKTOP`, and `IF_NOT_DESKTOP`. They are used in 75 out of 354 translation units. Therefore, enabling this feature flag potentially changes behavior of more than 20% of the system in a variety of locations that can not be easily traced by the developer. Our tool takes the affected macros into account automatically and visualizes the influence of the feature flag on the structure with just one click.

Figure 2 shows the structure of the translation unit `find.c` with three different configurations. Our tool improves the developers understanding of the resulting structur and behavior by making the commonalities and differences explicit.

6 DISCUSSION

With our prototype we focus on the visualization design to support developers when exploring software systems which are making use of conditional compilation. We have therefore placed more emphasis on usability than on feature completeness. The application scenario demonstrates how the visualization supports the developer's usual workflow and eliminates time-consuming steps. Q2 can already be answered completely. Q1 can only be answered partially since not all feature locations are visually detectable. The necessary information is already available, but is not yet accessible in the user interface. For example, when selecting a feature, it would be possible to highlight all methods affected by the feature. It would also be helpful to support saving and loading configurations for subsequent analyses as well as comparing complete configurations visually. As soon as these features are implemented, we will compare our tool with existing solutions.

As for many software visualizations scalability is a critical point and necessary to use the tool in practice. The generation process took one day on a conventional notebook. We can visualize systems with up to four million lines of code without any problems. If the visualization becomes too complex, performance may decrease. However, the visualization framework still offers a lot of potential to improve performance to visualize larger systems.



(a) All `find` specific features are enabled

(b) All `find` specific features are enabled except `FEATURE_FIND_EXEC` and `FEATURE_FIND_XDEV`

(c) All `find` specific features are disabled

Figure 2: Visualizing the structure of BusyBox's "find.c" with three different configurations

7 CONCLUSION

Our tool supports the developer to explore variability implemented with CPP, especially in the context of SPLs. It simplifices tracing and understanding the effect of enabling or disabling these flags with respect to the code compiled subsequently. Thus, it bridges the gap between feature models and diagrams as more abstract representations of variability and its concrete implementation with the means of CPP. However, some features are still missing for use in practice that need to be addressed in future work.

REFERENCES

[1] Greg J. Badros and David Notkin. 2000. A framework for preprocessor-aware C source code analyses. *Software: Practice and Experience* 30, 8 (2000), 907–924.

[2] David Baum, Jens Dietrich, Craig Anslow, and Richard Müller. 2018. Visualising Design Erosion : How Big Balls of Mud are Made. In *IEEE VISSOFT 2018*.

[3] David Baum, Jan Schilbach, Pascal Kovacs, Ulrich Eisenecker, and Richard Müller. 2017. GETAVIZ : Generating Structural, Behavioral, and Evolutionary Views of Software Systems for Empirical Evaluation. In *IEEE VISSOFT 2017*.

[4] Ira D. Baxter and Michael Mehlich. 2001. Preprocessor conditional removal by simple partial evaluation. In *Proceedings Eighth Working Conference on Reverse Engineering*. 281–290.

[5] Benjamin Behringer, Jochen Palz, and Thorsten Berger. 2017. PEoPL: projectional editing of product lines. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 563–574.

[6] Janet Feigenspan, Christian Kästner, Mathias Frisch, Raimund Dachselt, and Sven Apel. 2010. Visual support for understanding product lines. In *2010 IEEE 18th International Conference on Program Comprehension*. 34–35.

[7] Alejandra Garrido and Ralph Johnson. 2005. Analyzing multiple configurations of a C program. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. 379–388.

[8] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation - PLDI '12* (2012), 323. https://doi.org/10.1145/2254064.2254103

[9] Sheny Illescas, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2016. Towards visualization of feature interactions in software product lines. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*. 46–50.

[10] 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. *ACM SIGPLAN Notices* 46, 10 (2011), 805. https://doi.org/10.1145/2076021.2048128

[11] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A variability-aware module system. *ACM SIGPLAN Notices* 47, 10 (2012), 773. https://doi.org/10.1145/2398857.2384673

[12] Jacob Krüger, Ivonne Schröter, Andy Kenner, Christopher Kruczek, and Thomas Leich. 2016. FeatureCoPP: compositional annotations. (2016), 74–84. https://doi.org/10.1145/3001867.3001876

[13] Bernt Kullbach and Volker Riediger. 2001. Folding: An approach to enable program understanding of preprocessed languages. In *Proceedings Eighth Working Conference on Reverse Engineering*. 3–12.

[14] Panos E. Livadas and David T. Small. 1994. Understanding code containing preprocessor constructs. In *Proceedings 1994 IEEE 3rd Workshop on Program Comprehension-WPC'94*. 89–97.

[15] M. Vittek. 2003. Refactoring browser with preprocessor. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.* 101–110. https://doi.org/10.1109/CSMR.2003.1192417

[16] 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 *2014 Second IEEE Working Conference on Software Visualization*. 50–59.

[17] Flávio Medeiros, Thiago Lima, Francisco Dalton, Márcio Ribeiro, Rohit Gheyi, and B. ANDFONSECA. 2013. Colligens: A Tool to Support the Development of Preprocessor-Based Software Product Lines in C. In *Proc. áBrazilian Conf. áSoftware: Theory and Practice (CBSoft)*.

[18] 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 *Proceedings - 6th IEEE Working Conference on Software Visualization, VISSOFT 2018*. 107–111. https://doi.org/10.1109/VISSOFT.2018.00019

[19] Richard Müller and Dirk Zeckzer. 2015. The Recursive Disk Metaphor – A Glyph-based Approach for Software Visualization. In *Proceedings of the 6th International Conference on Information Visualization Theory and Applications (IVAPP '15)*. SciTePress, Setúbal, 171–176. https://doi.org/10.5220/0005342701710176

[20] Daren Nestor, Steffen Thiel, Goetz Botterweck, Ciarán Cawley, and Patrick Healy. 2008. Applying visualisation techniques in software product lines. In *Proceedings of the 4th ACM symposium on Software visualization*. 175–184.

[21] Yoann Padioleau. 2009. Parsing C/C++ Code without Pre-processing. In *Compiler Construction*, Oege de Moor and Michael I Schwartzbach (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 109–125.

[22] Diomidis Spinellis. 2010. CScout: A refactoring browser for C. *Science of Computer Programming* 75, 4 (2010), 216–231.

[23] 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

[24] Simon Urli, Alexandre Bergel, Mireille Blay-Fornarino, Philippe Collet, and Sébastien Mosser. 2015. A visual support for decomposing complex feature models. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. 76–85.

[25] Daniel G. Waddington and Bin Yao. 2005. High-fidelity C/C++ code transformation. *Electronic Notes in Theoretical Computer Science* 141, 4 (2005), 35–56.

MetricHaven — More Than 23,000 Metrics for Measuring Quality Attributes of Software Product Lines

Sascha El-Sharkawy
University of Hildesheim,
Institute of Computer Science
Hildesheim, Germany
elscha@sse.uni-hildesheim.de

Adam Krafczyk
University of Hildesheim,
Institute of Computer Science
Hildesheim, Germany
adam@sse.uni-hildesheim.de

Klaus Schmid
University of Hildesheim,
Institute of Computer Science
Hildesheim, Germany
schmid@sse.uni-hildesheim.de

ABSTRACT

Variability-aware metrics are designed to measure qualitative aspects of software product lines. As we identified in a prior SLR [6], there exist already many metrics that address code or variability separately, while the combination of both has been less researched. MetricHaven fills this gap, as it extensively supports combining information from code files and variability models. Further, we also enable the combination of well established single system metrics with novel variability-aware metrics, going beyond existing variability-aware metrics. Our tool supports most prominent single system and variability-aware code metrics. We provide configuration support for already implemented metrics, resulting in 23,342 metric variations. Further, we present an abstract syntax tree developed for MetricHaven, that allows the realization of additional code metrics.

Tool: https://github.com/KernelHaven/MetricHaven
Video: https://youtu.be/vPEmD5Sr6gM

CCS CONCEPTS

• **General and reference → Metrics**; • **Software and its engineering → Software product lines**; *Automated static analysis.*

KEYWORDS

Software Product Lines, SPL, Metrics, Implementation, Variability Models, Feature Models

ACM Reference Format:
Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2019. MetricHaven — More Than 23,000 Metrics for Measuring Quality Attributes of Software Product Lines. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3307630.3342384

1 INTRODUCTION

In software engineering, metrics are an established approach to characterize properties of software [7]. However, variability management is a key part of Software Product Lines (SPLs), which is not covered by traditional metrics. The SPL research community developed variability-aware metrics to address this issue, which received increasing attention over the last decade [5]. While a few very specialized metrics have been integrated into SPL-specific IDEs, there is still a lack of publicly available metric tool suites. Here, we introduce MetricHaven, an extension of KernelHaven [12], for the flexible execution and combination of variability-aware code metrics.

MetricHaven provides support for the measurement of traditional code metrics that ignore variability, SPL metrics that measure only variability, and the combination of both. Further, we allow the flexible combination of variability-aware code metrics with feature metrics, e.g., to detect complexity that arises through the combination of the variability model and code artifacts. MetricHaven provides configuration support to select among 23,342 metric variations (as of spring 2019). This is achieved by a specialized Abstract Syntax Tree (AST), which covers relevant information for measuring this broad variety of metrics. Through the underlying architecture of KernelHaven, which is optimized for parallelization, MetricHaven is able to measure a small number of selected code metrics on the whole Linux Kernel in less than 5 minutes, while the computation of all 23,342 metric variations takes around 3,5 hours. This means that in average each metric requires less than 1 second to measure the complete source code of Linux.

We regard MetricHaven as an experimentation workbench [11] for the analysis of SPL metrics. The provided concepts support researchers in developing and evaluating new variability-aware metrics. More precisely, we plan to analyze the potential of the metrics for fault prediction, by applying machine learning on the produced data. Also, practitioners may benefit from MetricHaven as they can directly use the tool for the detection of code smells.

2 CONCEPT

MetricHaven supports metrics for single systems, variability-aware metrics for SPLs, and arbitrary combinations of both. Further, we allow the combination of feature metrics for variability-aware metrics, if they count the number of features as part of their computation. As a result, MetricHaven supports more than 23,000 metric variations. Below we list the most important design decisions of MetricHaven:

Decoupling of Parsers and Metrics. The underlying architecture of KernelHaven provides data models, which serve as input for different kind of analyses [12]. This allows reuse of extractors and data models for the realization of new analyses. For the implementation of MetricHaven, we could benefit from existing extraction capabilities for variability and build models. The extraction of the variability model and its data model required minor adaptations for storing hierarchy information, which serve as input for some of the feature metrics. Only new extractors are required for measuring product lines that use modeling techniques that are not supported so far.

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342384

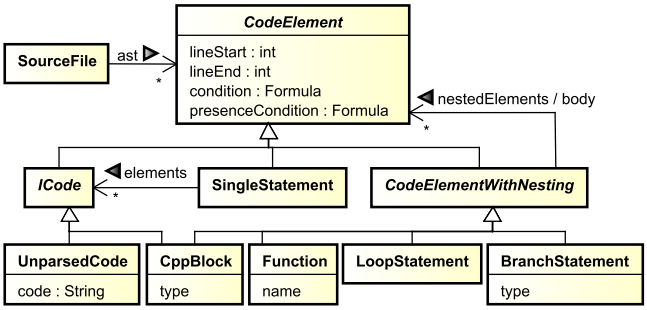


Figure 1: Excerpt of MetricHaven's AST.

Common AST. To allow the arbitrary implementation of single system and variability-aware code metrics, we require one common AST that stores elements of the annotation language, e.g., C-preprocessor, and elements of the programming language, e.g., C statements[1]. This is a challenging task, since the preprocessor is not part of the programming language and can be used at arbitrary positions inside a code file, independently of any syntax definitions. On the other side, we do not need a syntactical correct and complete AST, since the AST should not be used for compilation tasks or type checking analyses. Figure 1 presents an excerpt of our AST, which was motivated by our systematic literature study on variability-aware metrics [6] and an informal survey on traditional code metrics. The two most import elements are:

- The `CppBlock` is used to store preprocessor blocks (`#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`). Since it inherits from two classes (`ICode` and `CodeElementWithNesting`), it allows the insertion of preprocessor directives at arbitrary positions in our AST.
- The `SingleStatement` is the most fine-grained element of our AST, which is sufficient for the measurement of the most prominent metrics. Expressions are stored as unparsed elements.

Preprocessing Framework. The metric calculation operates on a per-function basis. This simplifies the implementation of metrics and also creates parallelization opportunities. However, some metrics (e.g. the Fan-In/-Out and Scattering Degree metrics) require a complete view on the whole code. For this, we implemented pre-processing components that run before the metric computation on the full models supplied by the extractors. The results of these pre-processing components are passed to the metric calculation component, which can look up these pre-calculated values.

2.1 Variability-aware Code Metrics

The AST given in Figure 1 stores enough information to realize traditional code metrics for single systems, variability-aware metrics for SPLs, and arbitrary combinations of both. Below, we present currently realized metrics, which may be selected by users to measure properties of SPLs (cf. Section 3). However, these metrics are not complete as the AST supports the definition of further metrics.

McCabe's Cyclomatic Complexity Measure [15] is a very common metric for single systems that computes the number of linear independent paths of the control flow representation. This may be computed by adding 1 to the sum of all branching statements (`if`, `while`, `for`, `case`, ...) [3]. This approach was also applied to variation points to measure complexity of variation points [14]. We support three alternatives to compute the cyclomatic complexity: The first counts only branching statements of the programming language. The second counts only variation points, e.g., `#ifdef`-blocks. The last one counts both kinds of branching statements, independently whether branching statements of the programming language are repeated in multiple variation points.

Nesting Depth (ND) measures the maximum / average nesting level of statements within control structures [3]. The authors argue that each nesting increases the complexity as a developer needs to consider all enclosing conditions when editing the code. This concept was also applied to variation points [13]. We support three variations of this metric by counting the nesting depth of statements of the programming language, variation points, and the combination of both.

Fan-In/-Out metrics are designed to trace how data is processed inside a program [3]. Ferreira et al. [8] designed metrics that measure the incoming and outgoing connections of functions in combination with the number of features used for the selection of these connections. Based on these metrics, we defined 3 variations of *Fan-In/-Out* metrics. The first variation measures only connections between functions and ignores the variability of the code. The second variation counts only conditional connections, that are function calls surrounded by at least one variation point inside the function. The third variation counts the number of features used to control the conditional inclusion of a function call.

Lines of Code (LoC) measures are very important metrics for single systems, but also in the context of SPLs. There exist a plethora of interpretations how to measure LoC. Jones [10] recommended to count statements instead of lines in order to have a formatting independent measure. *Lines of Feature code (LoF)* measures the lines of code that are controlled by a variation point [13]. In addition, the *Fraction of Annotated Lines of Code (PLoF)* computes $\frac{\text{LoF}}{\text{LoC}}$. We support LoC, LoF, and PLoF on statement level. Please note that the data model given in Figure 1 also conceptually supports to measure the lines of code instead of statements, as we also store the start and end line of each element.

Features per Function are designed to measure the complexity induced by the presence of `#ifdef`-blocks in the code [8]. We implemented 5 variations: 1.) Measures the number of features used inside a function. 2.) Measures the number of features that control the presence of the function, *excluding* the build model. 3.) Measures the number of features that control the presence of the function, *including* the build model. 4.) The union of the first and second metric. 5.) The union of the first and third metric.

Blocks per Function measure the number of variation points, instead of the features, used inside a code function [8]. The authors do not specify whether `#else`/`#elif`s are treated as separate blocks or as part of the `#ifdef`-block. For this reason, we implemented two variations of this metric: The first metric counts only the number of `#ifdef`-blocks and ignores its siblings. The second treats related `#else`/`#elif`s as separate blocks.

Tangling Degree (TD) counts the number of features used in variation point expressions [13]. We realized only one version, as we do not know about a relevant variation of it.

[1]The concepts we propose here, could also be applied well beyond C.

2.2 Feature Metrics

MetricHaven allows a flexible combination of code metrics with feature metrics. This can be done since most of the variability-aware code metrics count the number of features used to control the variation points. For these metrics we allow the usage of a feature metric to measure the complexity of the variation point instead of counting the number of used features only. Below, we introduce 7 feature metrics. Most of them are based on existing measures, but we developed also some new metrics for unmeasured properties.

Scattering Degree (SD) metrics are used to analyze whether a feature is implemented in a modularized way (low scattering) or is spread over the code base (high scattering) [17]. A high scattering indicates a fragile design and may significantly increase system maintenance. We re-implemnted two variations: SD_{VP}[13] is a commonly used metric that counts the number of variations points (i.e., `#ifdef`-blocks) a feature is used in. SD_{File}[9] is a more coarse-grained version as it counts the number of code files, in which a feature is used in at least one variation point.

Feature Size is inspired by the LoF-metric of the previous section. We count for each feature the statements controlled by that feature. We count a statement to all features that (in-)directly control the inclusion of the statement. For instance, the statements in Lines 3–5 of Figure 2 are controlled by the feature *A*. The idea of this metric is to provide an alternative to the previously defined scattering degree metric, which shows the impact of a feature to the implementation of the SPL.

The **Number of Constraints (NoC)** metric [1] is often used to measure the complexity of feature models, by counting the number of cross-tree constraints. We refined this metric and provide 3 feature metric derivations: NoC counts the number of all constraints of the feature model, where the respective feature is used in. Some modeling approaches allow a differentiation among constraints, e.g., the modeling approach of Linux requires that constraints are modeled as attributes of features [4]. We allow a separate analysis of constraints modeled as attributes of a feature (NoC_{out}) and constraints of other features referring to the measured feature (NoC_{in}).

The **Coefficient of Connectivity-Density (CoC)** is designed to measure how well graph elements are connected [1]. This is done by dividing the number of edges (parent child relations and constraints) by the number of features in a feature model. We modified this metric to operate on features instead of the complete feature model only and provide two alternative measures: The first alternative counts the number of non-transitive child features, to measure the number of parent-child edges. We do not count the parents, since it would increase all numbers by one only, excerpt for the root feature. The second alternative considers also all incoming and outgoing constraint connections, which are also measured by the NoC-metric of the previous Section.

The **Feature Types** metric is inspired by work that analyzes the amount of non-Boolean features used in publicly available SPLs [16]. We allow to specify weights for each feature type. This facilitates an analysis whether the usage of non-Boolean features in variation points increases the code complexity.

Feature Hierarchies is inspired by work on how structural properties impact the complexity of feature models. This includes the measurement of the depth of tree, the number of top features, or the number of leaf features [1]. We support two variations of hierarchy-based feature metrics: The first variation uses the nesting level of the feature as a feature metric. The second variation allows user-defined weights for top-level, intermediate, and leaf features.

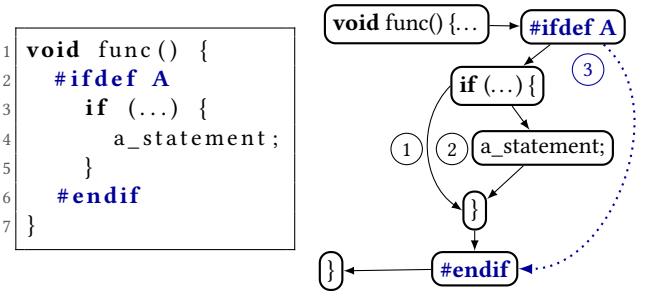


Figure 2: Code example and its variable control-flow.

This **Feature Locality** metric is a novel metric designed for the modeling behavior of Linux. Its variability model is spread over several hundred files [2]. The used include mechanism provides a modularization concept at which a sub feature-model can be stored at the same location as the realizing implementation. In addition, it is possible to provide alternative sub feature-models containing the same features with different constraints for different CPU architectures. We provide a feature metric to measure the distance of a feature realization and the feature definition. This is done by computing the distance in the file system of the sub feature model that defines the feature and the measured code artifact, which uses the respective feature. If the feature is defined by multiple alternative files, we count the shortest distance.

2.3 Example

Based on the variable control flow representation shown in Figure 2, we demonstrate how MetricHaven supports the computation of McCabe's cyclomatic complexity metric [15] as well as the variability-aware version of the metric [14]. Further, we demonstrate how to combine the variability-aware code metrics presented in Section 2.1 with feature metrics presented in Section 2.2.

McCabe on Code. McCabe's metric [15] computes the linear independent paths of the control flow representation of a program. Our example contains two linear independent paths, drawn with black lined arrows. This can be counted by adding 1 to the number of `Branch`- and `LoopStatements`.

McCabe on Variation Points. The variability-aware version of McCabe considers only paths created by variation points [14]. The example contains two paths. The black lined arrows are treated as one path (keeping the `#ifdef`-block) and the blue dotted arrows (removal of the `#ifdef`-block). This can be counted by adding 1 to the number of `CppBlocks`.

McCabe on Variation Points and Code. Further, MetricHaven allows a combination of the single system and the variability-aware code metric in a manner, which was not done before. For the combination of both metrics, we need to count all three paths of the control flow graph.

Combination with Feature Metrics. McCabe proposed to treat compound expressions in the form of Exp_1 `and/or` Exp_2 as separate paths as they could alternatively be realized with multiple if-then-else blocks [15]. Following his idea, we support measuring

of features in conditions rather than counting variation points only. Instead of adding 1 for each measured feature, we add the complexity value of a selected feature metric of Section 2.2. For example, we facilitate using the number of modeled constraints (NoC-metric) in which feature A appears in instead of counting the blue dotted path.

3 USAGE

MetricHaven is implemented as a plug-in for the KernelHaven infrastructure, which requires the Java runtime version 8 or higher. We recommend at least 16 GiB of RAM for the computation of all 23,342 metric variants. The default release of KernelHaven includes MetricHaven and required extractors for analyzing the Linux Kernel up to version 4.17. The variability and build model extractors work only on a Linux system with an installed GCC compiler plus the prerequisites for compiling the Linux Kernel. Other extractors may be supplied as additional plug-ins that allow analyzing other SPLs, which may not have these limitations.

An execution of MetricHaven requires a KernelHaven release[2] to be downloaded and unpacked. The release includes the `code_metrics.properties` configuration file, which can be used as a basis for the execution of MetricHaven. The KernelHaven process is started with the following command[3]:

```
java -jar KernelHaven.jar code_metrics.properties
```

KernelHaven's pipeline and the analysis components may be configured via Java properties files. The most relevant settings with respect to MetricHaven are:

- `source_tree` specifies the path to the SPL to analyze. For example, this can be the root directory of an unpacked Linux Kernel.
- By default, all 23,342 metric variations are executed. `metrics.code_metrics` may be used to select all variations of one metric. The provided configuration file contains a list of supported metrics. Further, `metrics.function_measures.all_variations` may be used to select a single metric variation.
- `code.extractor.threads` controls the number of threads used for parsing the source code of the product line.
- `metrics.max_parallel_threads` controls the number of threads used for calculating the metrics for a single function. This may be used for the computation of a large number of metric variations, otherwise the overhead of multithreading is too high.

4 CONCLUSION

We presented MetricHaven, an extension of KernelHaven for measuring metrics for single systems, variability-aware metrics for SPLs, and arbitrary combinations of both. Further, it supports the combination of feature metrics with variability-aware code metrics. Its flexibility allows the combination of 23,342 metric variations. This is achieved by an common AST that combines information of parsed code annotations (e.g., C-preprocessor) and the programming language (e.g., C-code).

MetricHaven is very fast. For example, it requires 3,5 hours for measuring all 23,342 metric variations on the complete Linux kernel[4]. This is about half a second per metric. Or, to put it differently, measuring 23,342 metrics, requires about 40ms per code function.

MetricHaven is a publicly available experimentation workbench designed to support researchers and practitioners in the analysis of annotation-based SPLs. New metrics may be developed based on the presented AST, without the need to develop new parsers. On the other side, the existing metrics may be applied without any changes to other imperative code files through the implementation of new extractors. This supports researchers in evaluating new SPL metrics. Also practitioners may benefit from MetricHaven as they can directly use the tool for the detection of code smells of annotation-based SPLs.

ACKNOWLEDGMENTS

This work is partially supported by the ITEA3 project REVaMP[2], 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.

REFERENCES

[1] Ebrahim Bagheri and Dragan Gasevic. 2011. Assessing the Maintainability of Software Product Line Feature Models Using Structural Metrics. *Software Quality Journal* 19 (Sep 2011), 579–612. Issue 3.
[2] 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 Trans. Software Eng.* 39, 12 (Dec 2013), 1611–1640.
[3] Samuel D. Conte, Herbert E. Dunsmore, and Vincent Y. Shen. 1986. *Software Engineering Metrics and Models.* Benjamin-Cummings Publishing Co., Inc.
[4] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the Kconfig Semantics and Its Analysis Tools. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2015).* 45–54.
[5] 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 (SPLC '17).* 19–28.
[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.
[7] Norman Fenton and James Bieman. 2014. *Software metrics: a rigorous and practical approach.* CRC Press.
[8] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. 2016. Do #Ifdefs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel. In *20th International Systems and Software Product Line Conference (SPLC '16).* 65–73.
[9] 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 (Apr 2016), 449–482. Issue 2.
[10] Capers Jones. 1986. *Programming Productivity.* McGraw–Hill, Inc.
[11] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. 2018. KernelHaven – An Experimentation Workbench for Analyzing Software Product Lines. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18).* 73–76.
[12] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. 2018. KernelHaven – An Open Infrastructure for Product Line Analysis. In *Proceedings of the 22nd International Systems and Software Product Line Conference — Volume 2 (SPLC '18).* 5–10.
[13] 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 *32nd ACM/IEEE International Conference on Software Engineering – Volume 1 (ICSE '10).* 105–114.
[14] Roberto E. Lopez-Herrejon and Salvador Trujillo. 2008. How complex is my Product Line? The case for Variation Point Metrics. In *Second International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'08).* 97–100.
[15] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Trans. Software Eng.* SE-2, 4 (1976), 308–320.
[16] Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2011. A Study of non-Boolean Constraints in Variability Models of an Embedded Operating System. In *15th International Software Product Line Conference, Volume 2 (SPLC '11).* 2:1–2:8.
[17] 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 Trans. Software Eng.* (2018), 1–1.

[2] https://github.com/KernelHaven/KernelHaven
[3] More details in appendix of https://sse.uni-hildesheim.de/en/research/publications/publikation-einzelansicht/?lsfid=41623&cHash=b244065d45b8c85e2cd7560559dda3dd
[4] Measurement of Linux version 4.15 on 2 Xeon E5-2650 v3 with 128 GB memory

Applying the QuARS Tool to Detect Variability

Alessandro Fantechi
Dipartimento di Ingegneria dell'Informazione,
Università di Firenze
Firenze, Italy
alessandro.fantechi@unifi.it

Stefania Gnesi
Istituto di Scienza e Tecnologie dell'Informazione "A.Faedo",
Consiglio Nazionale delle Ricerche,
ISTI-CNR
Pisa, Italy
stefania.gnesi@isti.cnr.it

Laura Semini
Dipartimento di Informatica,
Università di Pisa
Pisa, Italy
semini@di.unipi.it

ABSTRACT

In this demo paper we present how to use the QuARS tool to extract variability information from requirements documents. The main functionality of QuARS is to detect ambiguity in Natural Language (NL) requirement documents.

Ambiguity in requirements may be due to intentional or unintentional indication of possible variability; an ambiguity detecting tool can hence be useful to analysts and clients to figure the potential of a requirements document to describe a family of different products.

CCS CONCEPTS

• **Software and its engineering → Software product lines**;

KEYWORDS

Natural language requirements, ambiguity, variability

ACM Reference format:
Alessandro Fantechi, Stefania Gnesi, and Laura Semini. 2019. Applying the QuARS Tool to Detect Variability. In *Proceedings of 23rd International Systems and Software Product Line Conference - Volume B, Paris, France, September 9–13, 2019 (SPLC '19),* 4 pages.
https://doi.org/10.1145/3307630.3342388

1 INTRODUCTION

QuARS is a tool able to perform an analysis of Natural Language (NL) requirements in a systematic and automatic way by means of natural language processing techniques: the focus is on the detection of ambiguity defects, according to the process depicted in Figure 1, by identifying candidate defective words stored in dictionaries. In [1, 3] we have shown that ambiguity defects in requirements can in some cases give an indication of possible variability, either in design or in implementation choices or configurability aspects. In fact the ambiguity defects that are found in a requirement document may be due to intentional or unintentional references made in the requirements to issues that may be solved in different ways: the different implementations may give rise to a family of different products, instead of just a single product. We proposed a first classification of the forms of linguistic defects that indicate variation points, and we described a possible mapping from ambiguity or under-specification defects to fragments of feature models.

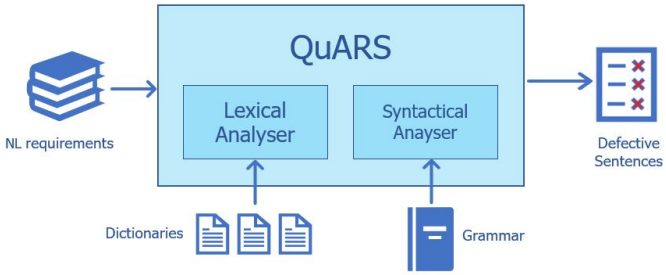


Figure 1: QuARS Process

We therefore have used the analysis ability of QuARS to elicit the potential variability hidden in a requirement document, with a two step process:

- A NL requirement document is given in input to QuARS to be analyzed according to its lexical and syntactical analysis engines, in order for identifying ambiguities.
- The detected ambiguities are analyzed in order to distinguish among false positives, real ambiguities, and variation points.

In [2] three requirement documents of medium to large size, publicly available and referring to different systems and domains, have been used to assess and validate the approach by using well known statistical measures. In this paper we illustrate as a demo how the above process is applied to detect variability in NL requirements using a simple case study. [1]

2 RUNNING EXAMPLE

The case study used as running example is a (simplified) e-shop, for which we consider the following requirements:

R1 The system shall enable a user to enter the search text on the screen.
R2 The system shall display all the matching products based on the search.
R3 The system possibly notifies with a pop-up the user when no matching product is found on the search.
R4 The system shall allow a user to create his profile and set his credentials.

[1] A video demo is available at https://fmt.isti.cnr.it/video.avi

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342388

R5 The system shall authenticate user credentials to enter the profile.
R6 The system shall display the list of active and/or the list of completed orders in the customer profile.
R7 The system shall maintain customer email information as a required part of customer profile.
R8 The system shall send an order confirmation to the user through email.
R9 The system shall allow a user to add and remove products in the shopping cart.
R10 The system shall display various shipping methods.
R11 The order shall be shipped to the client address or, if the "collect in-store" service is available, to an associated store.
R12 The system shall enable the user to select the shipping method.
R13 The system may display the current tracking information about the order.
R14 The system shall display the available payment methods.
R15 The system shall allow the user to select the payment method for order.
R16 After delivery, the system may enable the users to enter their reviews or ratings.
R17 In order to publish the feedback on the purchases, the system needs to collect both reviews and ratings.
R18 The "collect in-store" service excludes the tracking information service.

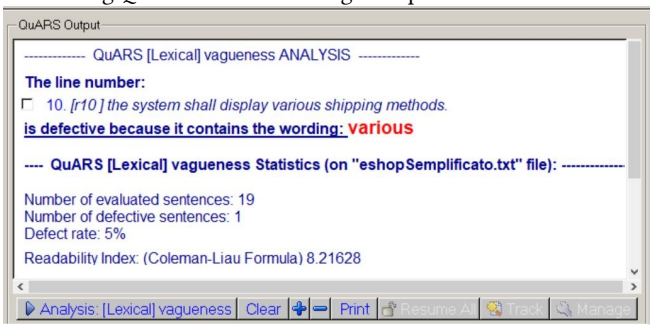
3 USING QUARS TO EXTRACT VARIABILITY

Looking at the set of requirements given in Sec.2 we can notice that there are a number of ambiguity defects, that are related to some words (the underlined ones) that introduce a possibility of different interpretation. In the remaining part of the section we will show the results of applying QuARS to this set of requirements to automatically detect ambiguity, and how these defects relate to variability and can be used to indicate variation points.

3.1 Vagueness

Vagueness means that the requirement contains words having no uniquely quantifiable meaning and therefore admits borderline cases, e.g., cases in which the truth value of the sentence cannot be decided. As an example: *the C code shall be* **clearly** *commented.* The QuARS Vagueness dictionary contains words like: *adequate, bad, clear, close, easy, far, fast, good, in front, near, recent, various, significant, slow, strong, suitable, useful, ...*

Running QuARS we find one vague requirement:



"Various" is a vague term, here indicating a variability about the different shipping methods that can be implemented.

In general, a vague word abstracts from a set of instances, that are considered as the "various" ones, and the process of requirement refinement will make these instances explicit. In terms of features, vagueness results in the introduction of an abstract feature (Fig. 2(a)). Once instances will be made explicit, they will be represented by sub-features, one for each instance.

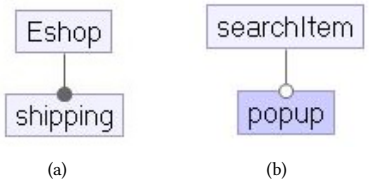
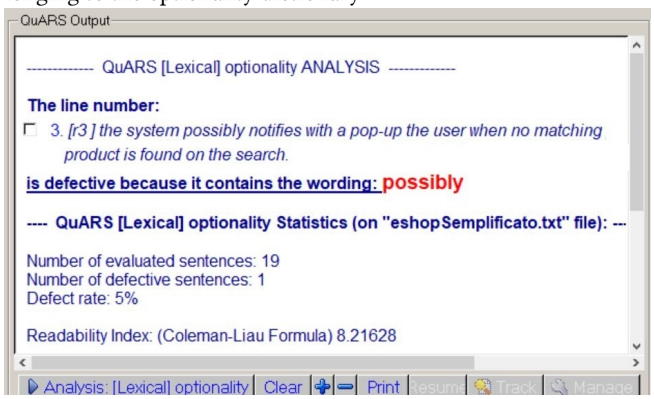


Figure 2: (a)-(b) Feature diagrams for vagueness and optionality

3.2 Optionality

Optionality occurs when a requirement contains an optional part, i.e. a part that can be considered or not: *the system shall be..,* **possibly** *without...* Example of Optionality-revealing words are: *possibly, eventually, in case, if possible, if appropriate, if needed, ...*

In our e-shop we find one requirements containing a term belonging to the optionality dictionary:



Optionality of a requirement is naturally expressed in a feature diagram with an optional feature. In the example, it has been recognized that the pop-up notification is an optional feature, as expressed by the fragment shown in Fig. 2(b).

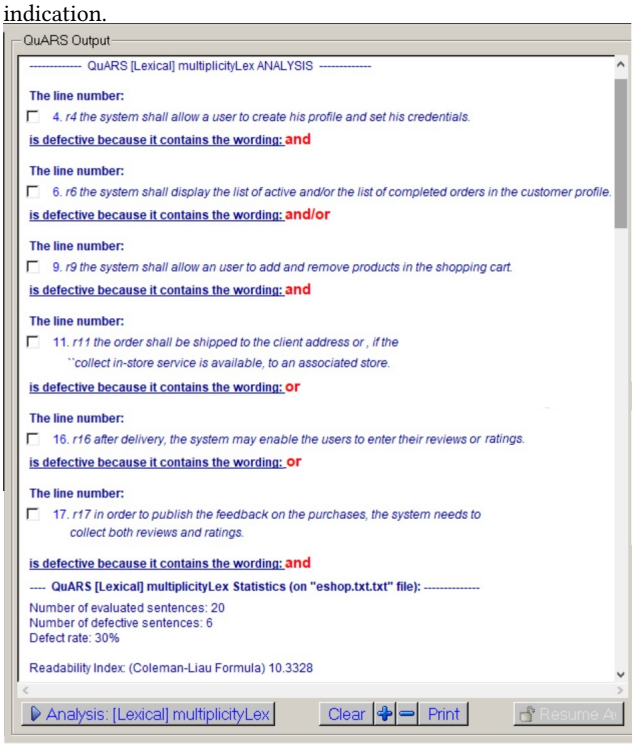
3.3 Multiplicity

Lexical multiplicity means that the requirements do not refer to a single object, but address a list of objects, using disjunctions or conjunctions (*or, and, and/or*), like in .. *opens doors* **and** *windows...* In particular, a requirement with an **or** leaves several possibilities open, and hence different products compliant to such requirement:

- **implicitly exclusive or:** the alternatives are mutually exclusive. In this case, the corresponding features are declared

as "alternative", with a partial diagram as the one used for vagueness.
- **weak or:** all the alternatives are optional.
- **logical** ∨**:** at least one of the alternatives should be present, but it is irrelevant which one.

Construct **and/or** is basically equivalent to the third interpretation of or, i.e. it requires that at least one of the alternatives should be present. The manual analysis can refine the interpretation to say which one of the alternatives is mandatory, and the other optional. An **and**, although recognized as a multiplicity, simply shows that all alternatives are mandatory hence it is not really a variability indication.



Requirement **R4**, matches exactly the case described above, conjunction of customer profile and credentials is represented by the fragment in Fig. 3(c). On the contrary, **R9** is a false positive. Including **and** in the multiplicity dictionary or not is a debatable issue: on the one side, it defines possible fragments of the feature diagram –a feature with two or more mandatory subfeatures–, on the other side, the analysis with QuARS may return many false positives. However, since adding or removing a word from a dictionary takes few seconds, the analyst can switch from one solution to the other.

The disjunction "rating or review" in **R16** is represented by the fragment in Fig. 3(b) in this case we used a **weak or** because of the weakness "may" (see section 3.4).

In the case of **R6**, the analyst has resolved the variability telling that the list of active orders should be mandatory, while the list of completed orders can be considered optional (see Fig. 3(a)).

Finally, **R11** defines the instances, "homeAddress" and "store" needed to make concrete the abstract feature "shipping" in Sect. 3.1: Their variability information is discussed in section 3.5. Requirement **R17** is dealt with in section 3.6.

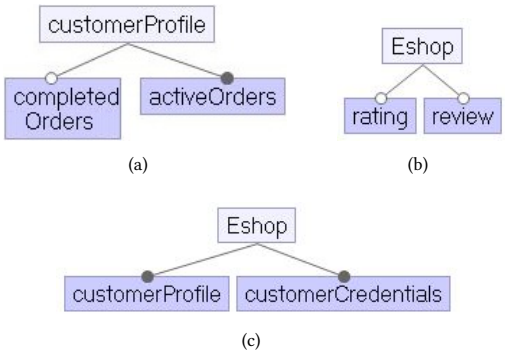
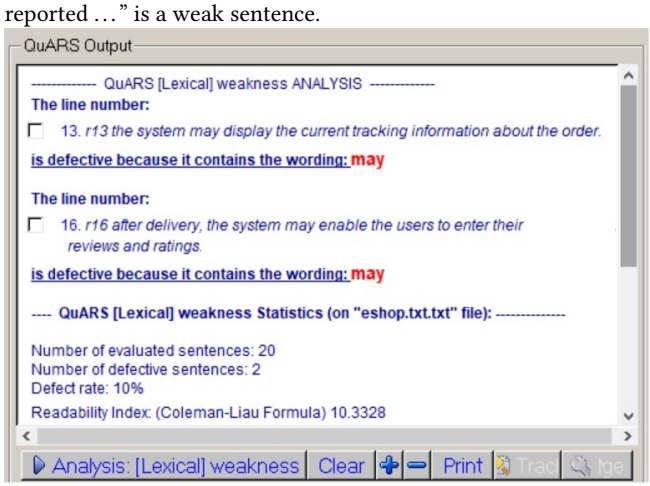


Figure 3: (a)-(c) Feature diagrams for multiplicity

3.4 Weakness

Weakness occurs when the sentence contains a "weak" verb. A verb that makes the sentence not imperative is considered weak (i.e. *can, could, may, ..*). For instance "the initialization checks ***may be*** reported . . . " is a weak sentence.



These two defective sentences clearly introduce an optionality, represented in Fig. 4(a) and 3(b), respectively.

Figure 4: (a) Feature diagram for weakness

3.5 Variability

In [2], we also exploited the capability of QuARS to add dictionaries for new indicators, namely variability related terms (*if, where, whether, when, choose, choice, implement, provide, available, feature, range, select, configurable, configurate*) and constraints identifiers (see Sect. 3.6).

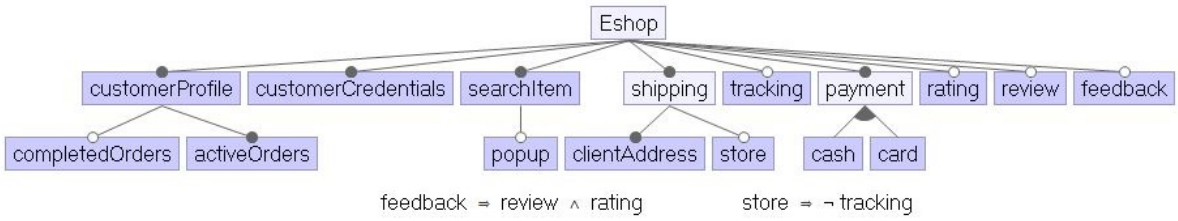
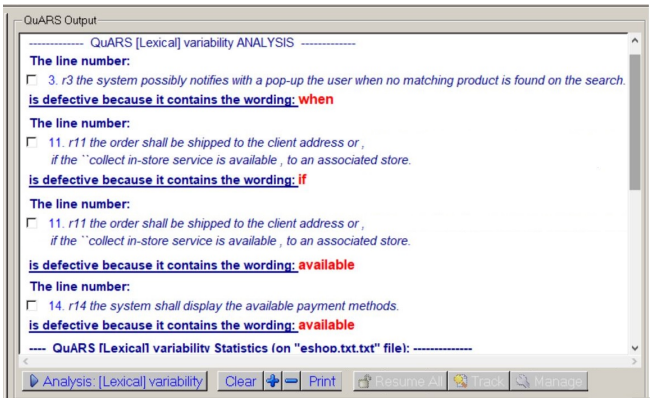
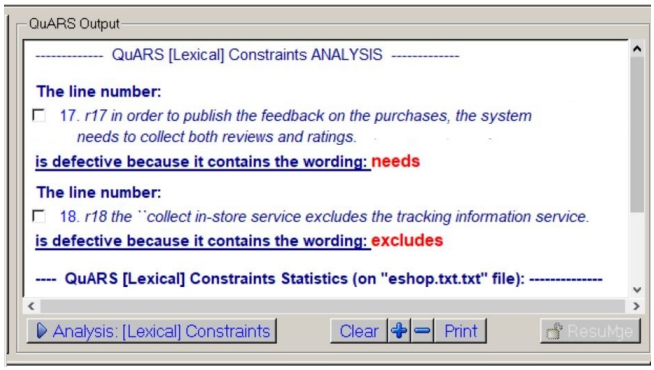


Figure 5: Feature model for e-shop

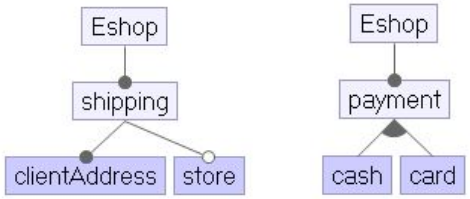




R3 is a false positive. **R11** and **R14** are actual variabilities. **R11** says that "ship to store" depends on the optional service "collect-to-store". Feature shipping is mandatory: at least a concrete subfeature must be present in each product. The choice, in this case, is to define "clientAddress" feature as mandatory, and "ship to store" optional, as expressed in the fragment of Fig. 6(a).

R14 is recognized to be a variability, that can be expressed by an *or* of the different payment methods that can be adopted for the system (see Fig.6(b)). Different payment methods were not specified in the original requirements, and have been expanded in this example as payment by card or payment by cash.

Notice that in both cases above the word "available" indicates a variability, but the nature of the variability is different because in the first case it is found inside an "if" context.



(a) (b)

Figure 6: (a)-(b) Feature diagrams for variability indicators

3.6 Constraints

Additional *cross-tree constraints* that is **requires**, indicating that the presence of one feature implies the presence of the other, and **excludes**, indicating that no system may contain the two features at the same time, may be detected looking at the occurrence of constraint-revealing terms such as: *expect, need, request, require, excludes* ....

In **R17** the term "needs" appears to indicate that a "publishFeedback" is an optional feature and that a *requires* cross-constraint has to be included in the model:

$$feedback \Rightarrow review \wedge ratings$$

Requirement **R18** adds another cross-constraint to the model due to term "excludes":

$$store \Rightarrow \neg tracking$$

Gluing together all the fragments extracted so far, we have manually built the feature diagram in Figure 5.

Acknowledgements

The work was supported by the Italian MURST PRIN project *IT MATTERS: Methods and Tools for Trustworthy Smart Systems* (2017FTXR7S).

REFERENCES

[1] A. Fantechi, A. Ferrari, S. Gnesi, and L. Semini. Hacking an ambiguity detection tool to extract variation points: an experience report. In R. Capilla, M. Lochau, and L. Fuentes, editors, *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS 2018, Madrid, Spain, February 7-9, 2018*, pages 43–50. ACM, 2018.

[2] A. Fantechi, A. Ferrari, S. Gnesi, and L. Semini. Requirement engineering of software product lines: Extracting variability using NLP. In G. Ruhe, W. Maalej, and D. Amyot, editors, *26th IEEE International Requirements Engineering Conference, RE 2018, Banff, AB, Canada, August 20-24, 2018*, pages 418–423. IEEE Computer Society, 2018.

[3] A. Fantechi, S. Gnesi, and L. Semini. Ambiguity defects as variation points in requirements. In *Proc. of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*, VAMOS '17, pages 13–19, New York, NY, USA, 2017. ACM.

RESDEC: Online Management Tool for Implementation Components Selection in Software Product Lines Using Recommender Systems

Jorge Rodas-Silva
University of Milagro
Milagro, Ecuador
jrodass@unemi.edu.ec

José A. Galindo
Jorge García-Gutiérrez
David Benavides
University of Seville
Seville, Spain
{jagalindo,benavides,jorgarcia}@us.es

ABSTRACT

Software product lines (SPL) management is one of the most important activities for the software engineer and it represents one of the key pieces of software product line engineering. When a software system grow fast, configuring a product becomes a costly and error-prone activity due to the amount of features available for configuration. This process becomes more complex when for each feature, there is more than one component that implements it. Currently the tools available for configuration management do not have automated mechanisms to facilitate the optimal components selection that meet the functions required by a given product. In this paper, we introduce a prototype component-based recommender system called RESDEC (REcommender System that suggest implementation Components from selecteD fEatures) designed to manage the best implementation components alternatives. Our tool is validated using WordPress-based websites where the implementation components are represented by plugins and the recommendations generated by RESDEC help interested parties in the search and efficient plugins selection to configure websites.

CCS CONCEPTS

• **Software and its engineering → Software development process management**; • **Software product lines**;

KEYWORDS

implementation components, software product lines, recommender systems, WordPress

ACM Reference Format:
Jorge Rodas-Silva, José A. Galindo, Jorge García-Gutiérrez, and David Benavides. 2019. RESDEC: Online Management Tool for Implementation Components Selection in Software Product Lines Using Recommender Systems. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3307630.3342390

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342390

1 INTRODUCTION

Although the process of creating new software has benefited from the functionalities availability in the form of components, obtaining an adequate configuration that meets a set of specific requirements is a complex activity. This is mainly due to the large number of options (specific application variants), which can be generated from the existing components combination. In an SPL, feature models capture the common and variable aspects of a family of similar products [2], which allow the generation of different software variants.

A common example of a software variant is a website designed in WordPress. Normally, a software developer manually searches for those components (plugins) that are feasible and most optimal for each website. This task takes time and does not always guarantee the selected components to be the most suitable (in terms of quality) for the required application. Two scenarios could arise during this configuration, on one hand, empirically selecting a component, in the practice, may not provide the expected results; and on the other hand, not having the criteria based on other users' experience regarding these components, could induce a bad selection and achieve a bad experience for the end user.

There are several tools in the literature to support the SPL configuration process [3, 10–14]. However, to the best of our knowledge and understanding, none of the tools developed so far supports the selection of implementation components when configure a product.

To overcome this problem, in this paper we introduce a component-based recommender system tool called RESDEC whose main focus is the use of explicit information that is generated by users about the implementation components and that is stored in an information repository.

The information provided by the users allows RESDEC to generate recommendations in real time through the use of recommender systems based on collaborative filtering and content filtering. In the practice, there are successful experiences with recommender systems, among which, online stores (Amazon), providers of state-of-the-art services (Netflix), among others stand out [1]. Taking the advantages offered by these systems in the context of selecting components to configure and customize software product lines is one of the main objectives pursued by RESDEC.

RESDEC supports a set of recommender algorithms based on the method described in [15] for the components selection. Our tool uses these algorithms in three different scenarios which originate

during the configuration of a website in WordPress: i) *Cold start*, which is executed when there is no information associated with the user, that is, when a user is going to set up a website for the first time. That is, this scenario recommends a list of plugins based on the trend or popularity; ii) *Recommendations of implementation components based on ratings*, which recommends plugins based on the experience of users with similar profiles according to the ratings; and iii) *Recommendations of implementation components based on features*, which recommends plugins based on contextual information about them, that is, the tags associated with the plugins used by users in the past.

2 RESDEC GENERAL OVERVIEW

RESDEC supplies users with information about which components are more suitable to use for SPL configuration. Next, we present the requirements and main elements of our tool.

2.1 Requirements

When managing an SPL of based-websites on WordPress, there is a large number of implementation components associated to features, such as payment options, shopping cart, security controls, among others. With a large number of features, managing the configuration of a web site on WordPress becomes a difficult task, even more difficult when, for each feature, there is more than one possibility of implementation. Therefore, implementing the website with a combination of appropriate components can be a complex activity to solve.

RESDEC uses feature models to describe the set of valid configurations. From these models, we can identify the implementation components associated with each feature and obtain the information that is developed around them. The processes for building these feature models are beyond the scope of this paper.

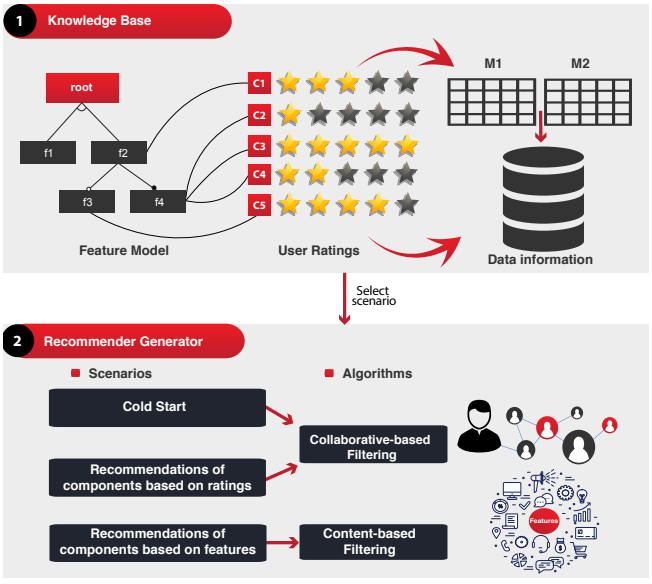


Figure 1: RESDEC elements

2.2 Elements

Fig. 1 illustrates the basic elements of our tool. As a first element, we have the *feature model* that describes the variability present in an SPL domain. From the feature model, it is possible to select a set of features that are associated to a series of components. Some of these implementation components contain information that is generated by users, such as, components ratings, report of bugs, number of installations, test cases executed, etc. In our case, we considered ratings that users make on WordPress plugins that have been used in website configuration in the past. The mapping of the feature model to the implementation components are described on the RESDEC website[1].

The information collected from the users allowed us to build the *Knowledge Base* (KB) composed of two matrices, $M1$ and $M2$. Matrix $M1$ relates user information, implementation components and ratings; while the matrix $M2$ relates the information of implementation components with their associated features.

After the knowledge base is built, we run the *Recommenders Generator*, which uses a set of algorithms that are commonly used in a Recommender Systems [4]. Specifically, we use collaborative-based and content-based recommender algorithms. Collaborative-based recommender systems [5, 7], also known as personalized recommender systems, are based on the analysis of user profiles, where recommendations are generated according to the tastes of users with similar preferences; while, content-based recommender systems [8] make recommendations based on the characteristics of the items without need to use information from other users.

These algorithms we have implemented in three different scenarios: *(i)* cold start; *(ii)* recommendations of implementation components based on ratings; and *(iii)* recommendations of implementation components based on features (see section 3.3).

For first scenario, we have implemented a classical popularity algorithm. While for the algorithms which run in the second scenario, we have employed Item-item KNN [16], User-user KNN [6] and the SVD factorization matrix [16]. Finally, for third scenario, we have implemented the algorithm TF-IDF [9].

In either case, regardless of the scenario and the selected algorithm, RESDEC recommends to the user a list of implementation components that could be used to configure a product in an SPL.

3 RESDEC TOOL SUITE

RESDEC offers two main functionalities: component repository management and automated analysis in the implementation components selection through recommender systems. The following are some advantages of RESDEC:

- It is easy adapt to any SPL configuration environment. To do this, the knowledge base has been designed based on three attributes commonly used by a recommender system (i.e users, items and ratings). This allowed us, that algorithms implemented in RESDEC receive these parameters as input and run without problems in any SPL scenario, for example: WordPress, Android, Mozilla, among others.
- It offers information about the implementation components and the ratings history made by the user.

[1]Real Case applied an eCommerce Site: http://resdec.com/about-case-study.html

– It provides a set of recommender algorithms that can be extended to provide better recommendation results in the three scenarios presented in this paper.
– It offers on screen, an updated history of the last components of implementation that have been of interest to the user.
– It allows obtaining recommendations, in execution time, of the most appropriate implementation components according to the feature selected by the user.
– It incorporates a case study based on a website software product lines that validates the scope of our tool.

3.1 Dataset

The dataset used by RESDEC has been built from CSV files. The data collected by these files corresponds to information extracted from WordPress. To obtain the data we have created a selenium based crawler[2]. First, we extract the list of plugins from WordPress[3], then, through the different plugins obtaining its number of stars, downloads, version, last update date, the WordPress version, the required PHP version, and associated tags. Finally, we obtain the list of users that reviewed the plugin, among its concrete review, and score. This information is then stored in a Json file[4] which is later exploited to generate the required inputs for RESDEC.

3.2 Architecture

RESDEC Tool has 3 components: a repository manager, a recommender manager and an output manager.

The *repository manager* responds to the requests of the stakeholders and structures the matrices $M1$ and $M2$ of the *DBKnowledge* (given in 2.2) through CSV's.

The *recommender manager* is in charge of processing the recommendations. It is developed in *Python* with a package of libraries that contain the algorithms that the recommender manager runs according to the scenario selected by the stakeholder.

For the *Cold Start* scenario RESDEC uses a classical popularity algorithm. While for the algorithms that run in the scenario *Recommendations of implementation components based on ratings*, employs the Scikit-surprise library[5]; and for the *Recommendations of implementation components based on features* scenario, it uses the Scikit-learn library[6].

The recommender manager is scalable and offers the possibility of implementing new recommender algorithms in any of the three scenarios presented in this paper.

The *output manager* interacts directly with the stakeholder using the repository manager and the recommender manager to generate the list of recommendations of the implementation components. It is designed in HTML5 and JavaScript, supported by the Semantic UI framework[7] used for the design of the interfaces. The interaction between the stakeholder and RESDEC is done through a web browser.

[2]Selenium WebSite: https://www.seleniumhq.org/
[3]WordPress plugins: https://wordpress.org/plugins/browse/popular/
[4]Json WebSite: https://www.json.org/
[5]Surprise website: http://surprise.readthedocs.io/en/stable/
[6]Scikit-learn website: http://scikit-learn.org/stable/index.html
[7]Semantic UI website: https://semantic-ui.com/

3.3 Web Application

To make our work accessible to the community, we present a RESDEC web application that eases the generation of recommendations to stakeholders that require guided assistance in the selection of plugins to configure a products line based on WordPress websites (see Fig. 2).

The main screen of RESDEC presents a menu with three recommendation scenarios where stackholders or users can configure an SPL, additionally incorporates a case study about eCommerce websites developed in WordPress (see Fig. **??** in Appendix B). RESDEC is available at *www.resdec.com*.

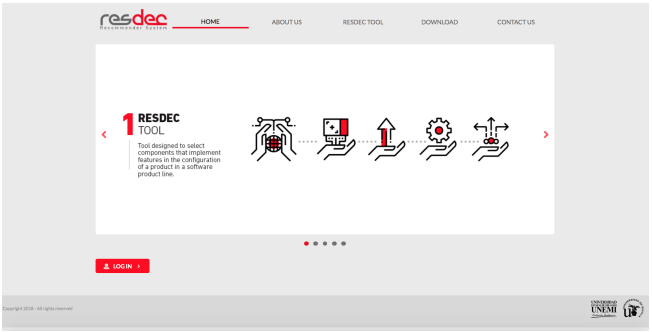


Figure 2: RESDEC web application

Next, we describe the different functionalities of our tool through an example based on the configuration of a tourism website in WordPress.

(1) The *Cold Start* option recommends components when there is no information associated with the user profile, i.e., when the user has no experience and is setting up a website for the first time. Suppose we are going to set up a new tourism website and we need to implement the *Social Media* function. In this case, the user selects the tag or tags associated with *Social Media* and specifies the number of desired recommendations. With the information provided by the user, RESDEC sets the recommendations based on the popularity of WordPress plugins and does not use the information associated with the user's profile (see Fig. **??** in Appendix B).
(2) The second option, *Recommendations of implementation components based on ratings*, from a component used by the user in previous configurations, recommends those components that users with similar profiles have used in the configuration of a product. Suppose that we are going to configure a *tourism website* that has already been implemented and in which we need new recommendation alternatives for *Social Media* function. In this case, the user selects the implemented plugin, *social-media-widget*, then specifies the number of desired recommendations and selects the algorithm to execute (SVD, item-item KNN or user-user KNN). With the information provided by the user, RESDEC establishes the recommendations based on the ratings that plugins similar to the one selected have been used by other users. In this scenario, RESDEC uses the information associated with the user's profile (see Fig. **??** in Appendix B).

(3) The third option, *Recommendations of implementation components based on features*, recommends implementation components based on the features of a component used by the user in previous configurations. That is, the list of recommendations is established based on the descriptive information of the components associated with the user profile. Suppose that we are going to configure a *tourism website* that has already been implemented and we need to replace or complement the feature *Social Media*, in this case, first the user selects the plugin implemented, *social-media-widget*, then the system will display the list of tags associated with the plugin through which it will establish recommendations. Then, we specify the number of desired recommendations and select the execution algorithm (TF-IDF). In this case, RESDEC establishes the recommendations based on the features, that is, on the similarity of the tags associated to the selected plugin with other plugins that use one or more of these tags. In this scenario, RESDEC also uses the information associated to the user's profile (see Fig. **??** in Appendix B).

(4) The option *Case Study applied to eCommerce Website*, shows a Feature Model that was built from information about websites designed in WordPress that are available on the Internet. This Feature Model has been implemented in an interactive way and describes the relations between features, the same ones that can appear when configuring an eCommerce website on this platform.

In the Feature Model, for example, by clicking on the *Shopping Cart* feature, the lower part of the model is configured for each scenario, showing only information associated with the selected feature. Thus, for scenario 1 it will show only the list of tags associated to that feature, in the same way in scenarios 2 and 3, it will display only the plugins that implement that feature. The recommendations in each scenario are executed in a similar way as described above (see Fig. **??** in Appendix B).

Along with the list of recommendations shown in the options described above, RESDEC shows a tab called *"You might also be interested"* that shows other components that might be of interest to the interest group. Finally, there is an option called *"About this case study"*, when you click on this option, it shows in detail the process that was carried out to build the feature model.

4 CONCLUSIONS

We have described in this paper a tool called RESDEC that uses a recommender system for the selection of implementation components that helps stakeholders or user to configure the features of an SPL.

The tool is based on a set of collaborative filtering and content filtering algorithms that are commonly used in recommender systems and that are executed in three possible scenarios that may arise when configuring an SPL. To demonstrate the scope of our tool, we have used real information extracted from WordPress (users, plugins, ratings and tags) that has allowed us to make a case study of the possible problems that a web developer may face when configuring a website on this platform. Specifically, we focus on recommender to users plugins that allow a website to be configured in an application domain. The future work focuses especially on the implementation of new recommender algorithms, experimentation and adaptation to other scenarios and contexts of practical relevance.

REFERENCES

[1] Jesús Bobadilla, Fernando Ortega, Antonio Hernando, and Abraham Gutiérrez. 2013. Recommender systems survey. *Knowledge-Based Systems* 46 (2013), 109–132.

[2] José A. Galindo, David Benavides, Pablo Trinidad, Antonio Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated analysis of feature models: Quo vadis? *Computing* 101, 5 (2019), 387–433. https://doi.org/10.1007/s00607-018-0646-1

[3] José A Galindo, Hamilton Turner, David Benavides, and Jules White. 2014. Testing variability-intensive systems using automated analysis: an application to Android. *Software Quality Journal* (2014), 1–41.

[4] David Goldberg, David Nichols, Brian M Oki, and Douglas Terry. 1992. Using collaborative filtering to weave an information tapestry. *Commun. ACM* 35, 12 (1992), 61–70.

[5] Joseph A Konstan, Bradley N Miller, David Maltz, Jonathan L Herlocker, Lee R Gordon, and John Riedl. 1997. GroupLens: applying collaborative filtering to Usenet news. *Commun. ACM* 40, 3 (1997), 77–87.

[6] Shyong K Lam and John Riedl. 2004. Shilling recommender systems for fun and profit. In *Proceedings of the 13th international conference on World Wide Web*. ACM, 393–402.

[7] Greg Linden, Brent Smith, and Jeremy York. 2003. Amazon. com recommendations: Item-to-item collaborative filtering. *Internet Computing, IEEE* 7, 1 (2003), 76–80.

[8] Michael Pazzani and Daniel Billsus. 1997. Learning and revising user profiles: The identification of interesting web sites. *Machine learning* 27, 3 (1997), 313–331.

[9] Michael J Pazzani and Daniel Billsus. 2007. Content-based recommendation systems. In *The adaptive web*. Springer, 325–341.

[10] Juliana Alves Pereira, Kattiana Constantino, and Eduardo Figueiredo. 2015. A systematic literature review of software product line management tools. In *International Conference on Software Reuse*. Springer, 73–89.

[11] 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. (2018).

[12] Juliana Alves Pereira, Pawel Matuszyk, Sebastian Krieter, Myra Spiliopoulou, and Gunter Saake. 2018. Personalized recommender systems for product-line configuration processes. *Computer Languages, Systems & Structures* (2018).

[13] Juliana Alves Pereira, Sandro Schulze, Eduardo Figueiredo, and Gunter Saake. 2018. N-dimensional Tensor Factorization for Self-configuration of Software Product Lines at Runtime. In *Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 1 (SPLC '18)*. ACM, New York, NY, USA, 87–97. https://doi.org/10.1145/3233027.3233039

[14] Juliana Alves Pereira, Sandro Schulze, Sebastian Krieter, Márcio Ribeiro, and Gunter Saake. 2018. A Context-Aware Recommender System for Extended Software Product Line Configurations. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 97–104.

[15] J. Rodas-Silva, J. A. Galindo, J. García-Gutiérrez, and D. Benavides. 2019. Selection of software product line implementation components using recommender systems: An application to Wordpress. *IEEE Access* (2019), 1–1. https://doi.org/10.1109/ACCESS.2019.2918469

[16] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*. ACM, 285–295.

Industrial Variant Management with pure::variants

Danilo Beuche
pure-systems GmbH
Magdeburg, Germany
danilo.beuche@pure-systems.com

ABSTRACT

The paper describes a demonstration of pure::variants, a commercial tool for variant and variability management for product lines. The demonstration shows how flexible product line (PL) architectures can be built, tested and maintained by using the modeling and integration capabilities provided by pure::variants. With pure::variants being available for a long time, the demonstration (and the paper) combines both basics of pure::variants, known to parts of the audience, and new capabilities, introduced within the last year.

CCS CONCEPTS

• **General and reference** → *Design*; • **Software and its engineering** → *Software product lines*;

KEYWORDS

Software Product Lines, Tools, Feature Modelling

ACM Reference Format:
Danilo Beuche. 2019. Industrial Variant Management with pure::variants. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, Article 4, 3 pages. https://doi.org/10.1145/3307630.3342391

1 INTRODUCTION

Product line engineering (PLE) can reduce the overall software production costs, but imposes extra complexity on the development process. Dealing with the commonalities and variabilities of the product variants and with the flexible software architectures makes PLE a real challenge.

The pure::variants approach supports handling of variability in all steps of product line engineering from requirements analysis to product generation. Extended feature models [1] are used for modeling of the problem domain. Family models are used to represent the variable architecture of PL solution domains. The pure::variants suite is intended to complement existing development and engineering tools with the necessary functionality to run a systematic reuse approach with PLE.

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342391

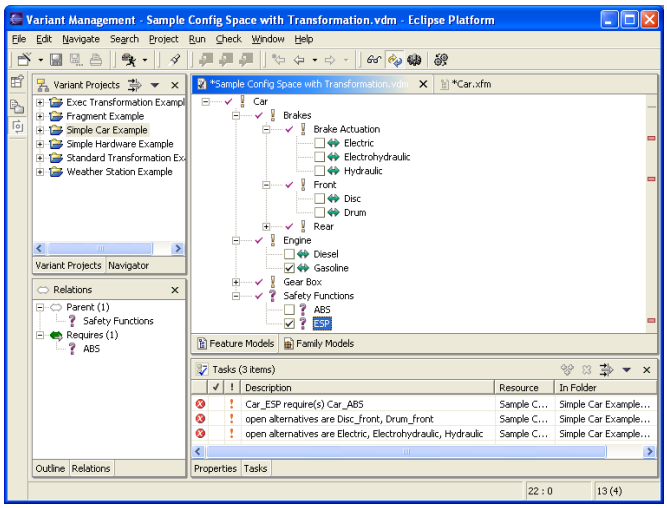


Figure 1: pure::variants Eclipse UI

pure::variants is being used in many different application domains such as car manufacturing, industrial automation, transportation, consumer tools and appliance production, communication technology, and more.

The demonstration shows how flexible product line (PL) architectures can be built by using the modeling capabilities provided by pure::variants. The main focus will be on recently and experimental capabilities of pure::variants.

The paper will give a brief overview about the basic principles pure::variants is built upon and will then describe the main elements of the demonstration.

2 TOOL SUPPORT FOR SOFTWARE PRODUCT LINES

The challenges of PLE involve high-level issues like development organization structure and processes, but also software architecture, design and implementation. One of the cross-cutting and most crucial tasks in PLE is the variant and variability management. However, most available software development tools do not offer support for explicit handling of this task.

Several important issues have to be considered for tools supporting the complete process of variability management:

- Easy, but universal model(s) for expressing variabilities and commonalities should be supported.
- Variability at all levels must be manageable.
- Introduction of new variability expression techniques should be possible and easy.

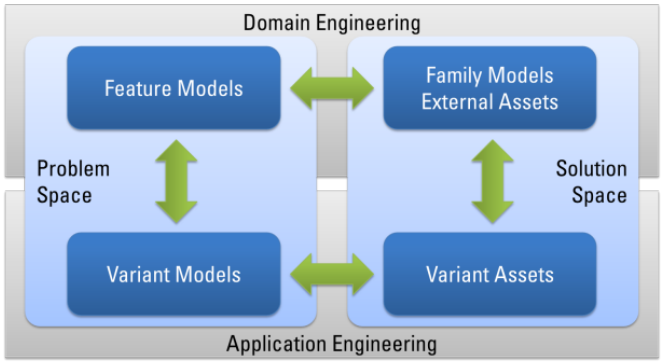


Figure 2: Mapping of pure::variants models to PLE activities

The pure::variants tool suite was developed to meet all these requirements. It is structured in a client/server architecture with an Eclipse plugin as its main graphical front end (figures 1, 4).

2.1 HOW PURE::VARIANTS WOKRS

The pure::variants-based tools are used in different phases of the product line development process. The development of product lines is basically divided into two main activities. One activity, the domain engineering is concerned with analysis of problem and solution domains, and with the identification and implementation of common and reusable assets. The second activity is concerned with deriving reusable assets, implementing product specific parts and testing of the individual products from the product line (application engineering). Both activities are heavily coupled, since changes introduced in one activity often impact the other. Tools like pure::variants provide the necessary tooling (in combination with the traditional development tools including version control systems and lifecycle management systems) to connect these activities successfully. Figure 2 shows the mapping of pure::variants Models and PL Assets to these product line activities.

Several model types are used to capture the information required to manage variability and variants on the different levels of domain knowledge, software design, and implementation. Feature models [1] play a key role in this. They allow a uniform representation of variabilities and commonalities of the products of the entire product line. Compared to original works on feature models, pure::variants supports an extended version of this concept, which not only provides Boolean features but also attributes with arbitrary types such as integer, float and string as well.

Attributes can be either of constant values or even automatically calculated as part of the variant evaluation. Rich constraint languages based on Prolog and OCL are provided to express complex relationships. pure::variants uses a declarative approach to describe variability and dependencies, which provides a good foundation for automated analysis and reasoning. pure::variants permits the definition of arbitrary attribute and element (element = features or family model elements, etc.) types. Implementations of the product line are described by family models. They enable the mapping of the problem space to variation points in the different solution space assets such as UML tools or requirements tools. This model type

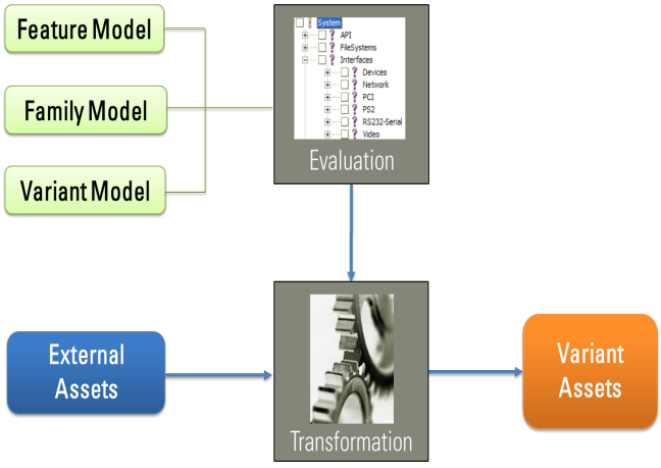


Figure 3: pure::variants data flow during variant derivation

was developed especially for the pure::variants technology, since existing modeling techniques such as UML were not suitable for this purpose. Family models are based on the same meta-model as pure::variants feature models and have thus the same expressiveness for variability. In essence, feature and family models are used in pure::variants to capture the domains knowledge regarding variability.

The variant model is used to describe an individual product. It describes the product's features and values associated with those features, and it is used to derive the final product from the family models.

Figure 3 depicts the simplified process of variant derivation with pure::variants. Most steps are performed automatically once the various models have been created. The product line developers have to provide feature models, family models, and the implementations themselves. To derive a specific variant features from the feature models have to be selected and possibly some associated values specified.

Once this variant model is successfully validated the transformation generates the variant realization (variant assets) from the variant result model using a product line specific transformation process which uses both, the pure::variants models, and the external assets to generate a set of variant specific assets.

3 INTEGRATION WITH 3RD PARTY TOOLS

An important element in PLE is the consistent representation of variability and variant information throughout the PL lifecycle. This requires in many cases exchange of data representing variability and variant related information between pure::variants and other tools (e.g. requirements management tools, modeling tools, code generators, version control systems) used by the different stakeholders. By providing specialized public extension APIs pure::variants makes this task very easy. For popular tools including DOORS and DOORS Next Generation, Enterprise Architect, Rhapsody, MagicDraw, EMF, Simulink, and others pure::variants provides such extensions out of the box. The interfaces for building extensions in pure::variants are open to everyone interested and can be (and

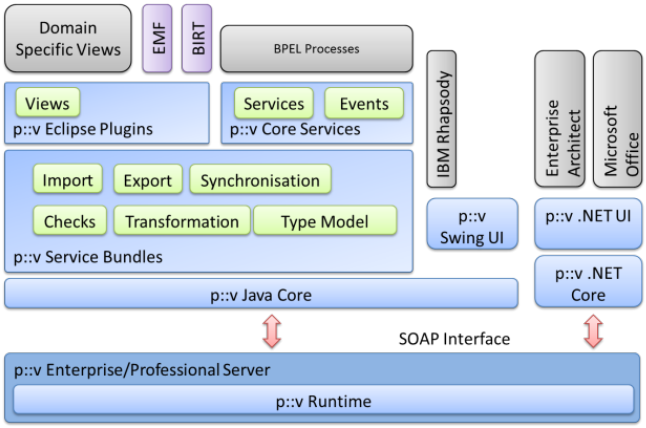


Figure 4: pure::variants building block overview

have been) used by others to integrate commercial and research tools with pure::variants [2].

The architecture of the pure::variants suite (Figure 4) shows its modular structure which is being used to build the different tools and integrations into 3rd party tools.

An important part of pure::variants Connectors with 3rd party tools is the integration of variability-related activities as seamlessly as possible by embedding relevant pure::variants functionality into the "standard" workflow of the tools. This means that variation points can be created and managed in the respective tool using tool-native UI elements if possible. We will demonstrate a new integration with a electronic design tool to show how commercial âĂIJnon-softwareâĂİ design tools can be coupled with feature models.

3.1 INTEGRATION CHALLENGES

A major challenge for any engineering organization is consistent configuration management across different tools / asset types often stored in multiple different version control systems or with internal tool-specific version control. We will demonstrate how our new capabilities for Global Configuration Management based on OSCL simplify product line and product evolution.

4 ADVANCED WORKFLOWS

Last but not least, the demonstration will show how pure::variants will in future simplify partial derivation for product line assets based on its already existing concepts for modeling variant configuration inheritance coupled with partial configuration aware asset transformations.

5 CONCLUSIONS

Crucial factors in the development of software product lines are adequate models for the domain analysis and design, programming languages that support the modular implementation even of crosscutting concerns, and an integration of all this in a user-friendly development environment, which supports the whole process. The demonstration will show that the pure::variants tool suite in combination with the Eclipse platform addresses these factors successfully. Further information on pure::variants including a free community edition can be found on [2].

ACKNOWLEDGMENTS

The work is supported by the Federal Ministry of Education and Research in project CrESt (funding id: 01|S16043N). The responsibility for the content rests with the authors.

REFERENCES

[1] 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 CMU/SEI-90-TR-21. Software Engineering Institute.

[2] pure-systems GmbH. 2019. pure::variants. Website. (2019). Available online at http://www.pure-systems.com/products/pure-variants-9.html; visited on July 2th, 2019.

Applying Domain-Specific Languages in Evolving Product Lines

Juha-Pekka Tolvanen
MetaCase
Jyväskylä, Finland
jpt@metacase.com

Steven Kelly
MetaCase
Jyväskylä, Finland
stevek@metacase.com

ABSTRACT

This demonstration shows how domain-specific languages for modeling and generating variant products can evolve together with the product line. In the demonstration, examples from practice are illustrated and executed, covering both domain engineering and application engineering. The examples cover the typical evolution scenarios: adding new features and variability points to a product line and then to existing products, changing their variation, and removing them completely from the product line. The evolution of the domain-specific languages, and the versioning of both the languages and products built with the languages, are demonstrated.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Domain specific languages**; *Software configuration management and version control systems*; Model-driven software engineering.

KEYWORDS

Software Product Line, product derivation and generation, Domain-Specific Language, Domain-Specific Modeling, MetaEdit+

ACM Reference Format:
Juha-Pekka Tolvanen and Steven Kelly. 2019. Applying Domain-Specific Languages in Evolving Product Lines. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3307630.3342389

1 INTRODUCTION

Product lines are constantly evolving: new features and their variability points will emerge and existing ones change or even become obsolete. At the same time, products that have already been delivered need to be maintained and updated with new features — including features not known when these products were originally delivered. Domain-Specific Languages (DSLs) can be used to cover the rich variation space within product lines [1]. With a DSL, the language definition itself defines the variation space and related rules. Language users then automatically follow the variation space when creating variant products. 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 [1, 5].

DSLs, and in particular related tools, have not always recognized the need for evolution nor provided tool support for it. In such cases, once a DSL has changed there has been a huge and often manual effort to update all the existing specifications to the new language version[1]. This unfortunate situation is the norm with textual DSLs and their syntax-oriented editors, but also found in many graphical modeling tools, including recently created ones (e.g. Eclipse-based [4]). In this demonstration we will show how a modern, mature tool, the MetaEdit+ language workbench, recognizes the importance of evolution and provides tool support for managing it. This demonstration focuses on features of MetaEdit+ 5.5 SR1.

2 METAEDIT+ FOR EVOLVING PRODUCT LINES

With MetaEdit+ Workbench [3], domain engineers define DSLs and get modeling tools for a specific product line in a fraction of the time needed by other tools [2]. First, experts define a domain-specific language as a metamodel containing the domain concepts and rules, and then specify the mapping from that to code by defining generators. DSLs can be collaboratively developed and maintained as well as versioned — using the functionality available in MetaEdit+. Language creation in an industrial setting takes on average 10 working days [7].

MetaEdit+ Modeler follows the defined modeling language and automatically provides full modeling tool functionality: editors, browsers, generators, collaborative model editing etc. Application engineers create variants by editing designs and by executing generators producing the product variants: their source code, configuration, deployment, tests, documentation etc. This leads to 5–10 times faster product derivation and time-to-market compared to traditional manual practices [6].

The use of DSLs, however, does not end once the first language version is created. Instead, DSLs need to be refined, enhanced and maintained along with the other assets of the product line. In particular, not only does the DSL (language and generators) evolve, but also changes to the language need to be reflected — ideally automatically and unobtrusively to language users — to the models specifying product variants. MetaEdit+ provides functionality to manage such evolution with its automatically collected history of models and all changes made to them. Both evolving DSLs and the variant specifications that have been created can be versioned to existing version control systems (e.g. Git or SVN). In the demonstration these tool features are shown in detail, following evolution scenarios from practice.

[1] One public example of this is the evolution of AUTOSAR metamodel.

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342389

3 CASES OF PRODUCT LINE EVOLUTION

In the demonstration we show the language definition and language usage scenario within different product lines, including Internet of Things applications, industrial automation systems and consumer products. The evolution of the language is addressed based on two dimensions:

(1) Nature of change: adding, changing or removing parts from the product line and thus from the DSL
(2) Whether the change influences only to the language definition, to the generator used for product derivation or to both.

Each case starts with presenting the results of domain engineering: DSL and generators. This is followed by showing DSL use in application engineering and generating product variants. Next, we present a scenario of product line evolution that calls for changes in the DSL. These changes are then discussed and implemented during the demonstration: changing the language elements, constraints, concrete syntax or generators.

Based on these language changes, a new version of the DSL is delivered to the language users, i.e. application engineers. With MetaEdit+ they can view the model history, inspect the changes made and version the product variants to version control systems like GitHub, Bitbucket etc. The same versioning approach is also available for domain engineers (for DSLs and related generators). The demonstration ends by showing the generated application code, its execution, or other artifacts relevant for the product variants developed.

REFERENCES

[1] Czarnecki, K., Eisenecker, U., Generative Programming, Methods, Tools, and Applications, Addison-Wesley, 2000.
[2] El Kouhen, A., Dumoulin, C., Gerard, S., Boulet, P., Evaluation of Modeling Tools Adaptation, 2012, https://hal.archives-ouvertes.fr/hal-00706701v2
[3] MetaCase, MetaEdit+ 5.5 User's Guides, 2017, https://www.metacase.com/support/55/manuals/
[4] Rocco Di, J., Ruscio Di, D., Narayanankutty, H., Pierantonio, A., Resilience in Sirius Editors: Understanding the Impact of Meta-Model Changes, Models and Evolution Workshop at Models Conference, 2018 (http://www.models-and-evolution.com/2018/papers/8.pdf)
[5] 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, Springer-Verlag, 2005.
[6] 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
[7] 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

Feature-Based Systems and Software Product Line Engineering with Gears from BigLever

Charles Krueger
BigLever Software, Inc.
10500 Laurel Hill Cove
Austin, Texas 78730 USA
+1 512 426 2227
ckrueger@biglever.com

Paul Clements
BigLever Software, Inc.
10500 Laurel Hill Cove
Austin, Texas 78730 USA
+1 512 994 9433
pclements@biglever.com

ABSTRACT

This paper describes a demonstration of the product line engineering tool and framework called Gears from BigLever Software. Gears is the automation at the heart of a *PLE Factory*, which itself is the conceptual construct at the heart of Feature-based Product Line Engineering. (Feature-based PLE is the subject of an upcoming ISO standard.) Gears provides the means to create and maintain a Feature Catalog via a unified feature modeling language; the means to create and maintain a Bill-of-Features Portfolio, which is a way to specify the members of the product line by the features that each one exhibits; and a single variation point mechanism that works in Shared Assets across the entire product lifecycle. The result is an automated production line capability that can quickly produce any product in the portfolio from the same, single set of shared assets.

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/10.1145/3307630.3342393

1 Tutorial Topic

Categories and Subject Descriptors

D.2.2 [**Design tools and techniques**]: *product line engineering, software product lines, feature modeling, hierarchical product lines, multistage configuration*

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342393.

General Terms

Management, Design, Economics.

Keywords

Product line engineering, software product lines, feature modeling, feature profiles, bill-of-features, hierarchical product lines, variation points, product baselines, product portfolio, product configurator, multistage configuration

1. INTRODUCTION

BigLever's Gears Systems and Software Product Line Engineering (PLE) Lifecycle Framework is the automation at the heart of a *PLE Factory*, which itself is the conceptual construct at the heart of Feature-based Product Line Engineering. (Feature-based PLE is the subject of an upcoming ISO standard.) Gears enables the integration of tools, assets, and processes across the full system and software product line engineering lifecycle and beyond. The Gears framework offers a unified PLE solution for requirements engineers, architects, modelers, developers, build engineers, calibrators, document writers, configuration managers, test engineers, project managers, and more. The Gears PLE Lifecycle Framework has integrations with many of the standard engineering tools in the industry. The PLE Lifecycle Framework and the accompanying methodology have enabled product engineering organizations such as General Motors, Lockheed Martin and the US Navy, General Dynamics and the US Army, and more – with some of the largest, most sophisticated and complex safety-critical systems ever built – to adopt the latest generation PLE approach. In fact, seven of the organizations in the SPLC Product Line Hall of Fame are users of BigLever's Gears solution.

In this demonstration, we look at Gears from two different perspectives: (1) from the point of view of the end user of a Gears-based unified PLE lifecycle solution, and (2) from the point of view of a tool maker integrating their tool into the Gears PLE Ecosystem.

With the Gears framework, product line development organizations have a common set of PLE concepts and constructs for all tools and assets, which allows traceability relationships and development processes to flow cleanly from one stage to another across the lifecycle:

2. NEW CAPABILITIES FOR 2019

New capabilities not previously demonstrated at SPLC include:

- An integration between Gears and PTC Windchill, expanding the footprint of Product Line Engineering

into the hardware-oriented world of Product Lifecycle Management (PLM), furthering the convergence of these two once disparate fields. Now, a user of Windchill can define feature-based variation points in the tool that Gears will be able to actuate to turn a BoM superset into a BoM for a specific product.

- A web-based version of Gears called Enterprise Gears, which makes all of the capabilities of Gears available via your favorite web browser.
- The ability to store and recall user-defined *attributes* with features and products descriptions.

3. DEMONSTRATION VIDEO

A comprehensive demonstration video can be found at http://vimeopro.com/user14048433/biglevervideos. On the page, scroll down to “BigLever Gears Product Line Engineering Lifecycle Framework Demo - Part 1.

Towards a Conceptual Model for Unifying Variability in Space and Time

Sofia Ananieva
FZI Research Center for Information Technology
Berlin, Germany
ananieva@fzi.de

Timo Kehrer
Humboldt University of Berlin
Berlin, Germany
timo.kehrer@informatik.hu-berlin.de

Heiko Klare
Karlsruhe Institute of Technology
Karlsruhe, Germany
heiko.klare@kit.edu

Anne Koziolek
Karlsruhe Institute of Technology
Karlsruhe, Germany
koziolek@kit.edu

Henrik Lönn
Volvo Group Trucks Technology
Gothenburg, Sweden
Henrik.Lonn@volvo.com

S. Ramesh
General Motors Global R & D
Warren, Michigan, USA
ramesh.s@gm.com

Andreas Burger
ABB Corporate Research Center Germany
Ladenburg, Germany
andreas.burger@de.abb.com

Gabriele Taentzer
University of Marburg
Marburg, Germany
taentzer@mathematik.uni-marburg.de

Bernhard Westfechtel
University of Bayreuth
Bayreuth, Germany
bernhard.westfechtel@uni-bayreuth.de

ABSTRACT

Effectively managing variability in space and time is among the main challenges of developing and maintaining large-scale yet long-living software-intensive systems. Over the last decades, two large research fields, Software Configuration Management (SCM) and Software Product Line Engineering (SPLE), have focused on version management and the systematic handling of variability, respectively. However, neither research community has been successful in producing unified management techniques that are effective in practice, and both communities have developed largely independently of each other. As a step towards overcoming this unfortunate situation, in this paper, we report on ongoing work on conceiving a conceptual yet integrated model of SCM and SPLE concepts, originating from a recent Dagstuhl seminar on the unification of version and variant management. Our goal is to provide discussion grounds for a wider exploration of a unified methodology supporting software evolution in both space and time.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; **Software product lines**; **Software version control**; *Abstraction, modeling and modularity.*

KEYWORDS

revision management, product lines, variability, version control

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342412

ACM Reference Format:
Sofia Ananieva, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, S. Ramesh, Andreas Burger, Gabriele Taentzer, and Bernhard Westfechtel. 2019. Towards a Conceptual Model for Unifying Variability in Space and Time. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3307630.3342412

1 INTRODUCTION

Complex software-intensive systems often need to exist in many variants in order to accommodate different requirements. At the same time, each of these variants is subject to continuous change and heavily evolves during all stages of software development and maintenance. Yet, software versions—resulting from evolution in time—and variants—resulting from evolution in space—are managed radically differently.

Version management, i.e., managing evolution in time, typically relies on a version control system, which provides basic storage services and supports intuitive workflows through a set of additional operations. The versioning of software artifacts has been extensively studied by the Software Configuration Management (SCM) research community since the advent of the first version control systems such as SCCS [22] and RCS [29] in the 1970s and 1980s. Research in this field has focused on versioning models, which define the artifacts to be versioned as well as the way in which these artifacts are organized, identified and composed to configurations [5]. Nowadays version control systems such as Subversion [20] or Git [15] are file-based, organizing versions of files in a directed acyclic version graph. Variants of a software artifact or an entire software system are represented by parallel development branches, where each of these branches has its own chronological evolution. While this strategy is simple, it does not scale in the case of multidimensional variability [5], and alternative version space organizations which tackle that problem never made it into mainstream [8]. Another issue is that the granularity provided, files,

is not always adequate when fine-grained development artifacts such as model elements or code sections are managed.

Next to traditional version management, the need for software mass-customization has been recognized within research on program families in the 1970s [17]. The field later evolved into software product line engineering (SPLE) [4, 7], which can nowadays be seen as the most successful approach to handling multidimensional variability in space, scaling up to several thousands of variants (a.k.a. products) of a software-intensive system. Instead of managing products as clones in parallel branches, SPLE advocates to create a product-line platform that integrates all the product features and contains explicit variation points realized using variability mechanisms such as conditional compilation or element exclusion. However, evolving product-line platforms over time is substantially more complex than evolving single variants [13].

In summary, SCM and SPLE are two widely established yet actively researched software engineering disciplines offering a variety of concepts to deal with software variability in time and space. However, neither research community has been successful in producing unified management techniques that are effective in practice, and both communities have developed largely independently of each other.

As a step towards overcoming this unfortunate situation, a recent Dagstuhl seminar on the unification of managing software evolution in time and space brought together leading practitioners and researchers from both disciplines to discuss each other's challenges, solutions, and experiences[1]. In this paper, we report on one of the results of the seminar, namely ongoing work on conceiving a conceptual yet integrated model of SCM and SPLE concepts. Clearly, both disciplines share a set of common concepts, notably the idea of composing a system from fragments which serve as units of versioning and as re-usable assets, respectively. These common concepts of system descriptions serve as a starting point for our conceptual model, before we will explore those concepts which we consider to be specific to one of the disciplines and give an idea of how those concepts could be combined for managing variability in time and space. Our goal is to provide discussion grounds for a wider exploration of a unified methodology supporting software evolution in both time and space. The value and possible usage scenarios of a conceptual model are twofold. It may be instantiated to characterize and classify existing approaches, to structure the state-of-the-art and to map and align both communities' core concepts. It may also pinpoint open issues and serve as a vehicle for evaluating different integration strategies on a high-level of abstraction.

2 CONCEPTUAL MODEL

In this section, we explain and illustrate basic design decisions of a conceptual model for unifying underlying concepts of SCM and SPLE. We assume that a software-intensive system is described as a set of different types of models. This includes all source code artifacts which are also considered as models of a particular type in the context of this paper.

In Figure 1, we present a basic conceptual model of variability in time and space. It is composed of three differently colored parts

[1] https://www.dagstuhl.de/19191

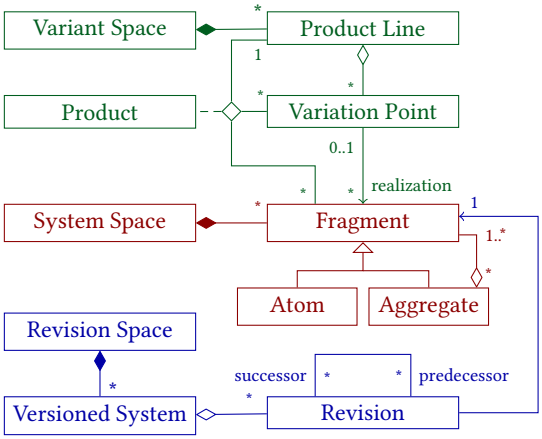


Figure 1: A Basic Conceptual Model of Variability in Time (blue), Variability in Space (green) and Shared Concepts (red)

corresponding to (i) concepts for variability in time (blue), (ii) concepts for variability in space (green), and (iii) concepts common to both (red). Figure 2 represents an extension to the introduced model and depicts the proposed integration of variability in space and time.

2.1 Common Concepts of System Descriptions

The common concepts of system descriptions represent the most fundamental intersection between basic concepts of SCM and SPLs. Our *System Space* of discourse is the set of all possible *Fragments* describing a software-intensive system, as depicted in Figure 1. Fragments are the essential concepts for defining a system description. A fragment can either be an *Atom* or an *Aggregate*. Depending on the concrete realization, such an atom can be on different levels of granularity, for example, a single character or a single file. An aggregate contains fragments on its own and may represent, for instance, the node of an abstract syntax tree. For the representation of fragments, we rely on a generalization of the composite design pattern by replacing the actual composite relationship with an aggregate one. Consequently, we do not enforce a hierarchy of containments but consider fragments to be composed to various combinations. The use of an aggregated representation allows us to form a complex structure of fragments serving as a description of a software-intensive system. The fragments contain enough information to allow the system to compose meaningful artifacts, for example a package structure that organizes a set of related classes.

2.2 Concepts for Variability in Time

The proposed model contains elements necessary for capturing the concepts for variability in time. In a general versioning approach, every element of the software system is under revision control and hence the concept of *Revision* is applied to each fragment in our model, as illustrated in Figure 1. Each revision references a particular fragment and explicitly represents the dynamic character of the fragment. A revision is intended to supersede its *predecessor*, e.g., due to a bug fix or refactoring. Thus, a sequence of revisions represents the chronological evolution of a fragment. To represent

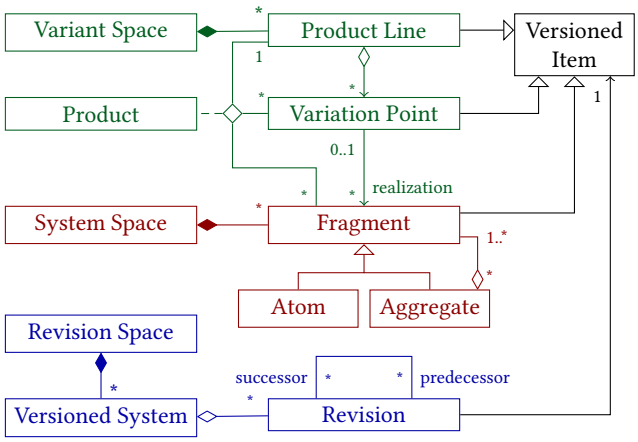


Figure 2: An Extended Conceptual Model (regarding Figure 1) for Combining Concepts of Variability in Space and Time

branching (which we consider a temporary divergence for concurrent development) along with merging, multiple (direct) *successors* and *predecessors* relate to a revision. This relation gives rise to a revision graph, which is a directed acyclic graph where each node represents a unique revision. The *Versioned System* is composed of revisions and represents the configurable space of a software system regarding temporal variability. The *Revision Space* is the set of all possible systems under revision control conceptually corresponding to the system space.

2.3 Concepts for Variability in Space

Next, we describe concepts for variability in space of the conceptual model. The entry point for this is the *Product Line*, depicted in Figure 1. The product line acts analogously to the *Versioned System* representing the configurable space of a system (or family of systems) regarding spatial variability. The *Variant Space* represents a set of all possible product lines. Product lines aim to systematically express the variability of its products in terms of an associated set of *Variation Points*. To allow reuse of variation points in multiple product lines, we consider the product line an aggregate for variation points. Each variation point has a set of configuration options where each option is realized by concrete fragments. A *Product* is considered fully specified in space if all existing variation points in the product line are bound to fragments, hence composing a complete product. A partial product, however, does not enforce the binding of every variation point. A ternary association represents the *configuration* of a product from a product line, which covers the selection of fragments that realize variation points. The association therefore describes the relationship between one product line and a selection of variation points and fragments. The product resulting from that configuration is denoted as an association class.

2.4 Combining Variability in Space and Time

So far, we have introduced generic concepts for variability in space and time along with shared concepts of system descriptions generally applied in SCM and SPLE. In the conceptual model depicted in

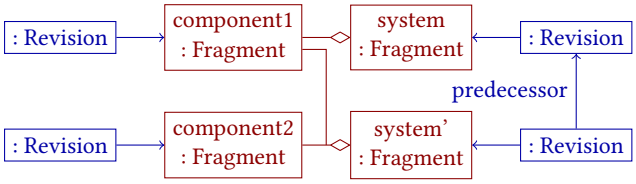


Figure 3: Exemplary Revision Model with two Revisions of a System and one Revision for each of the two Components

Figure 1, versioning has been applied only to the fragments but can be extended to concepts for variability in space to support effective evolution of variant-rich systems. For bridging the gap between variability in space and time and providing an integration of both, we propose an extended conceptual model along with the concept of a *Versioned Item*, illustrated in Figure 2. The versioned item represents a "higher-level" versioning of the introduced concepts by putting them under revision control. In this sense, the versioned item acts as a super class for the fragment, for the variation point and for the product line itself. Assuming that a product is fully derived from a product line, we refrain from versioning a product separately to avoid redundant revision control. In section 5, we present some of our main subjects to discussion during the development of both models.

3 APPLYING THE MODEL

In this section, we discuss some applications of the presented conceptual model for variability in space and time. We start with a small exemplary scenario that demonstrates how the model could be instantiated to represent revisions of a system. Next, since the model should be adapted and refined to be applied in actual approaches, we present different realization options. Finally, we discuss how appropriateness and expressiveness of the model can evaluated by applying it to existing approaches.

3.1 An Exemplary Scenario

Let us assume a simple scenario, in which an architectural model of software components is developed. Figure 3 shows the first two revisions of the system. In a first revision, a developer creates a composite fragment *system*, which contains an atomic fragment *component1* that represents an initial component. In a second revision, he or she adds a second atomic fragment *component2* that represents another component of the system, resulting in a modified system fragment *system'*.

In consequence, both atomic fragments exist in their initial revision, as atomic fragments may not be changed. The composite system fragment, in contrast, was updated, such that its second revision contains a revised representation of the initial revision referenced by the *predecessor* relation.

3.2 Realization Options

The presented conceptual model (Figure 1, Figure 2) is supposed to express the essential concepts of variability in space and time. It serves as a reference model and thus has to be adapted and refined when it shall be used to realize an approach for SPLE, SCM or a

combination of them. This means that the elements of the conceptual model can be mapped to different realization options. For example, the fragments referenced by revisions to describe the temporal development of a system may be realized as snapshots of the system (or parts of it), or they may be represented as deltas, which only describe differences between revisions. Such design decisions especially affect how a system can be derived from its variability and revision fragments, because delta-based approaches require to apply the complete history of deltas to derive a system state, whereas the system states are explicitly contained in a snapshot-based approach. Regarding variability in space, variable parts can be represented in terms of variation points, like in the orthogonal variability model (OVM) [21], or in terms of a hierarchic structure of features in a feature model [10].

As a general comment, some elements of the conceptual model may not be relevant when realizing an actual approach (such as for instance the System Space) but they however contribute to a complete conceptual system description.

3.3 Evaluation Plan

We can evaluate the expressiveness and appropriateness of our conceptual model by applying it to actual approaches that realize variability in space, in time or both. For example, transferring the revision concepts to the elements of a revision graph in Git and other revision control systems gives an indicator for the appropriateness of our revision concepts. This, for example, includes a mapping of fragments to delta descriptions and revisions to commits with a reference to the previous revision, commit message and a hash code for identification. For investigating the appropriateness of our revision concepts, we plan to transfer the description to Subversion, Git and EMFStore [12] as representatives of state-of-the-art approaches.

A common description of variability in space is achieved with feature models, which can, for example, be defined in FeatureIDE [11]. In the last years, several approaches that combine variability in space and time have been proposed, such as Ecco [9] and SuperMod [24], whereas others build on a delta-based representation of revisions and variability in space, such as DeltaEcore [25], SiPL [18, 19] and VaVe [1], or describe the evolution of SPLs systematically using model transformation rules [27]. If this conceptual model can be applied to several approaches that reflect the state-of-the-art for managing variability in space and time, we can assume generality of the model with high evidence. For that reason, apart from pure SCM or SPLE approaches, we will especially apply the conceptual model to the approaches that combine both.

4 RELATED WORK

There has been a considerable amount of work on concepts and terminology by both research disciplines of SCM and SPLE. For SCM [5, 16, 20], a prominent conceptual model to capture variability in time is represented by the *version model* describing diverse revision concepts such as the specification of objects to be versioned, revision identification or the supported graph topology, i.e., whether revisions are only structured in a linear sequence or if they form a directed acyclic graph that represents temporary development branches and merges of them. For SPLE [3, 21], a common conceptual model to capture variability in space is represented by the *variability model* which defines the variability of a product line, i.e., by introducing the concept of variation points and defining types of variation for a particular variation point. Compared to the conceptual model presented in this paper, there is however not yet a fundamental conceptual approach for variability in space and time that aims to integrate established concepts of both research disciplines (e.g., revisions and variation points) that is (i) declarative in nature by means of describing systems with variability in space and time and (ii) independent of realization by abstracting from the specification or tooling of systems with variability in space and time.

Conradi and Westfechtel [5] extend the notion of *version models* to represent the interplay between variability in space and time but concentrate on identifying several degrees of freedom for realizing both dimensions. Apart from the representation of fragments, this, for example, concerns versioning granularities or delta-based realization options such as *directed deltas* or *symmetric deltas*. The *Uniform Version Model (UVM)*, introduced by Westfechtel et al. [30], serves as a common model for basic SCM and SPLE concepts but is intertwined with realization aspects as it relies, for instance, on propositional logic and *selective deltas* (corresponding to *version identifiers* in SCM or *presence conditions* in SPLE in order to control visibility of fragments). Schwägerl [23] extends and specifies the UVM, among other things removing the concept of fragments and considering each element a *versioned item* which corresponds to our definition of it. In conclusion, we argue that conceptual models exist for either SCM or SPLE but not for both combining concepts for variability in space and time. If they do, however, they are intertwined with aspects of specification and thus are not declarative in the sense of the introduced conceptual model.

Variation Control Systems (VarCS) are actual approaches that integrate concepts of SCM and SPLE at different levels of granularity. Linsbauer et al. [14] provide a classification of VarCS and compare selected systems such as *SuperMod* and *Ecco*. In subsection 3.3, we refer to some of the actual solutions which potentially represent realizations of the conceptual model. As depicted in the evaluation plan, this requires future investigation.

From a software language engineering perspective, Architecture Description Languages like EAST-ADL [6] and AUTOSAR [26] support variablility in space, but a combined concept of variants and revisions is usually left to tool implementations or delegated to a SCM solution.

Finally, following the same goal but having a focus different from ours, another paper which emerged from the same Dagstuhl seminar 19191 studies potential synergies and combinations of product-line analyses (i.e., analyses to cope with variability in space) and regression analyses (i.e., analyses to cope with variability in time) [28].

5 DISCUSSION & FUTURE WORK

In this paper, we have proposed a conceptual model that describes the essential concepts of modeling variability of a software system in space and time. We have also presented an extended model that unifies those concepts to represent revisions of variable system parts. To validate that our model is *general* and *appropriate* in the

sense that we are able to map its elements to actual approaches for describing such variability, we will apply the model to existing approaches, such as Ecco, SuperMod or DeltaEcore in future work.

Several design decisions in the conceptual model were subject to intensive discussion and may be validated when transferring the model to actual approaches. One central subject of discussion is whether branches in revision control systems are a concept of variability in time to support temporary divergence for concurrent development, or whether they represent a realization of variability in space, as they support the existence of products at the same point in time. For the time being, we chose to follow the former notion and allow branches in the revision concepts, but appropriateness of that decision has to be validated in future work. Another subject of discussion which requires future validation is whether or not to consider the product a subclass of the *versioned item*. According to Antkiewicz et al. [2], product derivation is either fully automated or followed by manual post-processing (corresponding to the so-called *governance levels* L5 and L6). In the case of fully automated product derivation (L6), a product represents a fully derived artifact for which revision control becomes superfluous since the product line is already put under revision control in the extended model. When manual post-processing takes place (L5), a product does not represent a fully derived artifact anymore for which revision control becomes reasonable again. Additionally, the semantics of several concepts is only defined through the mechanisms that operate on them. For example, the configuration of a product from a product line, variation points and fragments is expressed in our model, but constraints that define which variation points and fragments may be selected have to be ensured by a configuration mechanism. The same applies to the unifying concept of our extended model. To define what the relations between revisions of product lines, variation points and fragments are, a mechanism that defines how they can be combined has to be defined. Designing such a mechanism, based on the presented model, should be the next step towards a unifying concept for variability in space and time.

ACKNOWLEDGMENTS

We thank all participants of Dagstuhl-Seminar 19191 (Software Evolution in Time and Space: Unifying Version and Variability Management) for the discussions and especially Thorsten Berger and Lukas Linsbauer for their contributions in the breakout group.

REFERENCES

[1] Sofia Ananieva, Heiko Klare, Erik Burger, and Ralf Reussner. 2018. Variants and Versions Management for Models with Integrated Consistency Preservation. In *Proc. Intl. Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 3–10.

[2] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Stefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. 2014. Flexible Product Line Engineering with a Virtual Platform. In *Proc. Intl. Conference on Software Engineering*. ACM, 532–535.

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer.

[4] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

[5] Reidar Conradi and Bernhard Westfechtel. 1998. Version Models for Software Configuration Management. *ACM Comput. Surv.* 30, 2 (June 1998), 232–282.

[6] Philippe Cuenot, Patrick Frey, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Mark-Oliver Reiser, Anders Sandberg, David Servat, Ramin Tavakoli Kolagari, Martin Törngren, and Matthias Weber. 2010. The EAST-ADL Architecture Description Language for Automotive Embedded Software. In *Model-Based Engineering of Embedded Real-Time Systems: International Dagstuhl Workshop. Revised Selected Papers*, Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz (Eds.). Springer, 297–307.

[7] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

[8] Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. 2005. Impact of Software Engineering Research on the Practice of Software Configuration Management. *ACM Transactions on Software Engineering and Methodology* 14, 4 (2005), 383–430.

[9] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *Proc. Intl. Conference on Software Engineering*, Vol. 2. IEEE, 665–668.

[10] 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 University Software Engineering Institute.

[11] 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 *Proc. Intl. Conference on Software Engineering*. IEEE Computer Society, 611–614.

[12] Maximilian Koegel and Jonas Helming. 2010. EMFStore: a Model Repository for EMF models. In *Proc. Intl. Conference on Software Engineering*, Vol. 2. ACM, 307–308.

[13] 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.

[14] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *Proc. Intl. Conference on Generative Programming: Concepts & Experience*. ACM, 49–62.

[15] Jon Loeliger and Matthew McCullough. 2012. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc.

[16] Stephen A. MacKay. 1995. The State of the Art in Concurrent, Distributed Configuration Management. In *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops on Software Configuration Management*. Springer, 180–193.

[17] David Lorge Parnas. 1976. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering* SE-2, 1 (1976), 1–9.

[18] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. 2015. SiPL–A Delta-Based Modeling Framework for Software Product Line Engineering. In *Proc. Intl. Conference on Automated Software Engineering*. IEEE Computer Society, 852–857.

[19] Christopher Pietsch, Udo Kelter, Timo Kehrer, and Christoph Seidl. 2019. Formal Foundations for Analyzing and Refactoring Delta-Oriented Model-Based Software Product Lines. In *Proc. Intl. Systems and Software Product Line Conference*. ACM.

[20] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. 2008. *Version Control with Subversion: Next Generation Open Source Version Control*. O'Reilly Media, Inc.

[21] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[22] Marc J. Rochkind. 1975. The source code control system. *IEEE Transactions on Software Engineering* 1, 4 (1975), 364–370.

[23] Felix Schwägerl. 2018. *Version Control and Product Lines in Model-Driven Software Engineering*. Ph.D. Dissertation. University of Bayreuth, Bayreuth.

[24] Felix Schwägerl and Bernhard Westfechtel. 2016. SuperMod: Tool Support for Collaborative Filtered Model-driven Software Product Line Engineering. In *Proc. Intl. Conference on Automated Software Engineering*. ACM, 822–827.

[25] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. Integrated Management of Variability in Space and Time in Software Families. In *Proc. Intl. Software Product Line Conference*. ACM, 22–31.

[26] Miroslaw Staron and Darko Durisic. 2017. *AUTOSAR Standard*. Springer, 81–116.

[27] Gabriele Taentzer, Rick Salay, Daniel Strüber, and Marsha Chechik. 2017. Transformations of Software Product Lines: A Generalizing Framework Based on Category Theory. In *Intl. Conf. on Model Driven Engineering Languages and Systems*. IEEE Computer Society, 101–111.

[28] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards Efficient Analysis of Variation in Time and Space. In *Proc. Intl. Workshop on Variability Modelling of Software-Intensive Systems*. ACM.

[29] Walter F. Tichy. 1982. Design, Implementation, and Evaluation of a Revision Control System. In *Proc. Intl. Conference on Software Engineering*. IEEE Computer Society, 58–67.

[30] Bernhard Westfechtel, Bjørn P. Munch, and Reidar Conradi. 2001. A Layered Architecture for Uniform Version Management. *IEEE Trans. Softw. Eng.* 27, 12 (Dec. 2001), 1111–1133.

Towards Modeling Variability of Products, Processes and Resources in Cyber-Physical Production Systems Engineering

Kristof Meixner
Christian Doppler Lab CDL-SQI, ISE
TU Wien, Austria
kristof.meixner@tuwien.ac.at

Rick Rabiser
Christian Doppler Lab MEVSS, ISSE
Johannes Kepler Univ. Linz, Austria
rick.rabiser@jku.at

Stefan Biffl
Inst. of Information Systems Eng.
TU Wien, Austria
stefan.biffl@tuwien.ac.at

ABSTRACT

Planning and developing *Cyber-Physical Production Systems* (CPPS) are multi-disciplinary engineering activities that rely on effective and efficient knowledge exchange for better collaboration between engineers of different disciplines. The *Product-Process-Resource* (PPR) approach allows modeling products produced by industrial processes using specific production resources. In practice, a CPPS manufactures a portfolio of product type variants, i.e., a product line. Therefore, engineers need to create and maintain several PPR models to cover PPR variants and their evolving versions. In this paper, we detail a representative use case, identify challenges for using *Variability Modeling* (VM) methods to describe and manage PPR variants, and present a first solution approach based on cooperation with domain experts at an industry partner, a system integrator of automation for high-performance CPPS. We conclude that integrating basic variability concepts into PPR models is a promising first step and describe our further research plans to support PPR VM in CPPS.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

Variability Modelling, Product-Process-Resource, Cyber-Physical Production System

ACM Reference Format:
Kristof Meixner, Rick Rabiser, and Stefan Biffl. 2019. Towards Modeling Variability of Products, Processes and Resources in Cyber-Physical Production Systems Engineering. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3307630.3342411

1 INTRODUCTION

In recent years, computation and communication technologies increasingly collaborate with connected smart physical devices, building ubiquitous *Cyber-Physical Systems (CPSs)*, which are capable of autonomously interacting with their environments, including humans [33, 41]. *Cyber-Physical Production Systems (CPPSs)*, such as automated car manufacturing plants or steel mills, reflect the characteristics of *CPSs* to industrialized manufacturing [6, 33]. Planning and developing successful *CPPSs* are engineering tasks requiring a multi-disciplinary team effort of engineers from different domains, such as mechanical, electrical and software engineering [6, 8]. Innately, the involved disciplines establish different mindsets resulting in a variety of heterogeneous engineering artifacts. Therefore, in multi-disciplinary teams a proficient knowledge exchange is crucial for an effective and efficient collaboration, subsequently requiring an adequate knowledge representation, which often does not exist.

Model-based engineering artifact representations can help to bridge the gap of engineering knowledge transfer [4, 59] as they are easy to exchange among domains and can be transformed to represent relevant concepts of other domains. The *Product-Process-Resource (PPR)* approach [48], for example, allows defining relations between products to produce (e.g., a cake or rocker switch), the associated production processes (e.g., assembling or welding), and the necessary production resources (e.g., machines or robots). The *Formal Process Description (FPD)* [56] is a formal notation to realize *PPR* by modeling production processes with their input, intermediate, and output products as well as the resources needed in the processes to manipulate these products. Such concepts allow modeling, e.g., *assembly sequences*, which define the manufacturing steps for a particular product variant as a basis for designing the layout of the *CPPS* and describing/optimizing production plans, easily.

However, (a) approaches, such as the *FPD*, are still limited for modeling variability of the involved artifacts and (b) existing *Variability Modeling (VM)* approaches [11] might not be able to deal with the variability of processes, products, as well as resources in a *CPPS* context. Existing work from the area of *Software Product Line (SPL)* engineering either addressed (software) product variability [2, 38, 55] OR process variability [45, 52], but not their combination. Integrated software process and product lines have been proposed as a vision [43], but, at least in the *CPPS* context, this vision has not been achieved. Further, production resources have been only indirectly addressed in work on non-functional properties and product lines [51].

A combined approach for *PPR Variability Modeling* (*VM*) seems to be not readily available. While a multi-product line approach [21] might seem promising on the first glance, we will show that product, process, and resource variability should be modeled in an integrated manner, and not as separate product lines. Finally, in a *CPPS* context, integrated *PPR* modeling is not the only challenge for a *VM* approach, primarily due to the heterogeneity of artifacts and involved domains, and due to the continuous and independent evolution of products, processes, and resources.

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342411

In this paper, we identify and detail the challenges for *VM* approaches in the *CPPS* context. Furthermore, we evaluate a first *PPR* notation extension for variability in a case study with domain experts at an industry partner and present a research agenda directed towards achieving the goal of modeling the *PPR* variability in *CPPSs*.

The remainder of this paper is structured as follows. Section 2 presents the background of *CPPS* and *PPR* and discusses related work on *PPR* and *VM*. Section 3 describes our research questions. Section 4 presents an illustrative use case, which is used to derive challenges for *VM* in *CPPS* presented in Section 5. Section 6 presents a first solution approach and describes a research agenda. We conclude the paper in Section 7.

2 BACKGROUND AND RELATED WORK

This section summarizes related work on *CPPSs*, *PPR*, and *VM*.

2.1 Cyber-Physical Production Systems

The scientific community uses several definitions of *CPSs*, which Gunes et al. [20] summarize in a survey on *CPSs* concepts and challenges as *"complex, multi-disciplinary, physically-aware next-generation engineered systems that integrate embedded computing technology into the physical phenomena"*. Examples range from home automation solutions based, e.g., on Google Echo, over truck fleet logistics using GPS for real-time coordination, to large-scale infrastructure like smart grids allocating resources depending on energy load. Key capabilities of *CPSs* are, e.g., robustness, safety, and adaptation to environmental characteristics in real-time through, e.g., self-diagnosis, self-adaptation, and self-maintenance [33, 41]. Krüger et al. [27] state that a central challenge of *CPSs* stems from managing the variability of their heterogeneous aspects.

In this paper, we utilize [6] that defines *CPPSs* as advanced production systems building the foundation for the 4th industrial revolution [33]. Using smart physical infrastructures, the latest data, computer, and communication technology, as well as modern production methods, like additive manufacturing, *CPPSs* facilitate optimized production processes along the value chain for a variety of products with a broad range of characteristics.

As mentioned *CPPS* engineering requires efforts of engineers from various domains, building a multi-disciplinary environment [6]. Furthermore, engineers often build groups, e.g., basic vs. detailed planning, depending on the process phase [23]. On top, they consult experts with cross-cutting knowledge, like safety and security, for support. In such settings, disciplines employ different paradigms and use heterogeneous engineering artifacts, technologies, and tools [34]. For instance, electrical engineers use wiring plans as *CPPS* perspective and specific tools to manipulate them. Therefore, engineers working together necessarily establish auxiliary common concepts [35] that describe similar elements of *CPPSs* and act as interfaces between domains. For example, in wiring plans control lines for sensor inputs are modeled and connected via wiring to robot sensors that are familiar to mechanical engineers. However, due to the varying *CPPS* perspectives and the heterogeneous artifacts, auxiliary concepts are often insufficient. Aiming at enabling more effective and efficient knowledge exchange, it is, therefore, crucial to establish representations that include relevant *CPPSs* aspects and their commonalities and variability.

2.2 Product-Process-Resource

CPPS engineering is often optimized for *intra-disciplinary* processes [23]. Still, the heterogeneity and incompatibility of paradigms, domain-specific tools, and artifacts in the *interdisciplinary* knowledge exchange may result in low-quality data and, subsequently, in planning errors.

On top, the internal semantics of engineering artifacts, such as *Excel* spreadsheets, which are frequently used, often require expert interpretation. Therefore, model-based, machine-readable, and easily exchangeable engineering representations are the foundation for bridging gaps in the knowledge transfer [4] by providing common concepts [35] between engineering domains. However, as the purpose of *CPPSs* is to manufacture products by combining production resources in production processes [15], these varying aspects are inseparably linked. This implies the need for comprehensive models to express the requirements of products towards *CPPSs*.

Schleipen et al. [48] coined the term *PPR model* based on the trinity of (a) the product with its characteristics and components (Bill of Materials), (b) the processes with their characteristics (Bill of Operation), and (c) the resources executing the processes. In the *PPR* model, these aspects of a *CPPS* are treated as first-class objects.

Several approaches support modeling *PPR* concepts, like the initially proposed approach of Schleipen et al. [48], which heavily builds upon *AutomationML*[1] and the *ISA 95* [22], which indirectly allows the representation of *PPR* but is more oriented at batch processing and the description of interfaces between manufacturing information systems. We decided to use the *FPD* approach [56] that, in contrast to the before mentioned approaches, provides a direct, tool- and technology-agnostic way to represent *PPR* concepts. The *FPD* defines a graphical notation and a data model for modeling connected production processes, including their boundaries that accept process input products and yield output products by employing specific resources (see left side of Figure 1 for an example).

2.3 Variability Modeling

There is a plethora of *systematic (literature) reviews (SLRs)*, mapping studies, and surveys that discuss *Variability Modeling* (*VM*) approaches [3, 5, 10, 11, 18, 39]. For example, in a tertiary study Raatikainen et al. [39] investigate structured reviews in the field of software product lines to extract (a) how software variability is modeled, (b) which kinds of variability models exist, and (c) which level of evidence was found in the reviews. Their findings show that research on the variability of process models and quality attributes is underrepresented. Rather than new approaches of *VM*, they advocate the adaptation and combination of existing approaches to particular problem contexts to make *VM* applicable for industry.

Galster et al. [18] provide an *SLR* investigating current trends of *VM* and how variability is handled in software engineering, but also gaps in the research area. Their results show that only a few works consider business processes as artifacts in *VM* and that design-time qualities, including evolvability, has been less addressed in existing research.

There is some work focusing on *process variability/software process lines*. For example, Rosa et al. [44] report on a survey on *VM* for business processes. They found variability mostly modeled by

[1] AutomationML - https://www.automationml.org/

extending existing business process models with *VM* elements, e.g., by exchanging process tasks for sub-process templates. Classical *VM* techniques, such as feature or decision models, were more seen as decision support during customization when choosing elements to add to the business process models. Simmonds et al. [52] describe their experiences of using a mega-modeling approach to define processes and their variability. Rouillé et al. [45] propose an approach to apply the *Common Variability Language* to model requirements variability and their relation to development processes. Lamprecht et al. [28] present a behavior-oriented approach to variability management for supporting process developers in selecting processes that match service constraints.

There is also some work on *non-functional properties and variability*, which might be useful to deal with resource variability [51]. Sincero et al. [53] describe their *Feedback Approach*, which extends traditional *SPL* engineering to improve the configuration of non-functional properties. Ghezzi and Sharifloo [19] explain how probabilistic model checking techniques and tools can help verify non-functional properties of configurations derived from a *SPL*.

There is a rich body of work on modeling *CPSs* [12, 29], e.g., modeling architecture and behavior [42] or modeling system goals of self-adaptive systems. Modeling *CPSs* clearly has been recognized as an important approach to deal with their complexity and heterogeneity [29]. Variability, however, has not been the scope of these works. Except for some initial works [27] that mainly discuss challenges and potential solutions, to the best of our knowledge, there is no approach explicitly and systematically tackling the variability of products, processes, and resources in an integrated manner in the context of *CPPSs*. Existing work from the *SPL* community, e.g., on partial models or perspectives on feature models [26, 49] and on supporting interdisciplinary product lines [16, 24], needs to be adapted and extended for this context.

3 RESEARCH QUESTIONS

This section raises research questions that we identified from related work and discussions with domain experts at an industry partner – a provider of automation solutions for high-performance *CPPSs*.

***RQ1**: Which aspects of CPPS engineering mainly challenge the capabilities of existing variability modeling methods? CPPS* engineering is embedded in a multi-disciplinary environment with diverging characteristics to software engineering, where *SPL* engineering stems from, and traditional manufacturing. To enable *VM* in the context of *CPPS* engineering in practice, we need to identify which particular aspects of such projects challenge existing *VM* approaches. We address this *RQ* by analyzing the literature from Section 2 and interviewing domain experts of our industry partner. By answering *RQ1*, we aim to sketch and investigate possible research directions to address the identified challenges and adapt existing *VM* approaches for use in the *CPPS* context.

***RQ2**: How can a PPR model, such as the FPD, be extended to represent both product and process variability as foundation for co-evolution of products and processes in the design of assembly sequences in CPPS engineering? FPD* is a standard approach that reflects *PPR* aspects of *CPPS* engineering. However, the approach does not take *VM* aspects into account, e.g., to model the commonalities and variability of products and related manufacturing processes. Therefore, we investigate how extending the *FPD* modeling notation can represent both solution space variability and problem space variability to domain experts as a foundation for designing co-evolution of products and processes in the design of assembly sequences in *CPPS* engineering. A first goal is enabling engineers to model variability in the *assembly sequence* efficiently based on a suitable notation as a fully combined variability model is likely to overwhelm them. Despite this goal we need to further investigate how to combine the *PPR* notation with advanced aspects of *VM*.

4 ILLUSTRATIVE USE CASE

We present the *cake baking* use case, based on [60], to illustrate the variability in *CPPSs* and challenges for existing approaches.

Figure 1 shows, on the left-hand side, a part of a *FPD* modeling a *cake baking production process* along with its products and resources, and, on the right-hand side, corresponding orthogonal feature models of the particular *PPR* aspects. Exemplary variation points from the *FPD* model, indicated with gray circles and an ID *VarX*, are reflected in the orthogonal feature models.

A cake consists of a varying number of layers, a filling between the layers and an optional gloss. The layers require three input factors, i.e., *Flour*, *Milk*, and *Eggs*, depicted as circles in the figure, which together create a *Raw Dough* after a mixing process (*Mix ingredients*), shown as a rectangle, using a *Mixer*, illustrated as rounded rectangle. The dashed line around the mentioned *PPR* aspects shows the boundary of the process step. In a further process (*Bake dough*), the *Raw Dough* is then baked in an oven to create a *Cake Layer*.

Quite similar to the previous two process steps, the process *Cook filling* requires two ingredients to create a cake filling. However, in this case we can choose from alternative input factors, i.e., the *Fruit*, marked here for better readability with *Var1* in the *FPD* of Figure 1. Corresponding to the *FPD*, on the top right-hand side of the figure there is a feature model describing the *products*, i.e., the *cake* and its parts. *Fruit* are modeled as an alternative feature and marked as *Var1* in the model. Depending on the type of *Fruit* used, the *Filling* needs to be cooked at different temperatures levels. Therefore, two further variation points are highlighted in the figure. First, tagged with *Var2*, the energy of the *process*, added as *Heat*, needs to be adjusted in a certain range. Second, the *resource* used to process the *Fruit*, i.e., the *Stirrer*, labeled with *Var3* in the *FPD*, needs to fulfill the requirements of the *Filling* to be cooked. These variation points imply *cross-model dependencies* between the variability models and features that are illustrated as the red arrows. In our case, depending on the chosen feature for *Fruit*, the process requires appropriate heat implying in the *Resource Variability* feature model, to choose the *Hot Stirrer* when the cake should contain a *Strawberry Filling*.

An additional variability, indicating cross-model dependency, is the decision whether or not to put a *Gloss* on the cake (*Var4* in the figure). If the optional feature *Gloss* is not selected, the whole process (*Create gloss*) with its products and resources, indicated by the dashed boundary, has to be neglected. Further, more complex constraints that can be modeled at design time or run time could

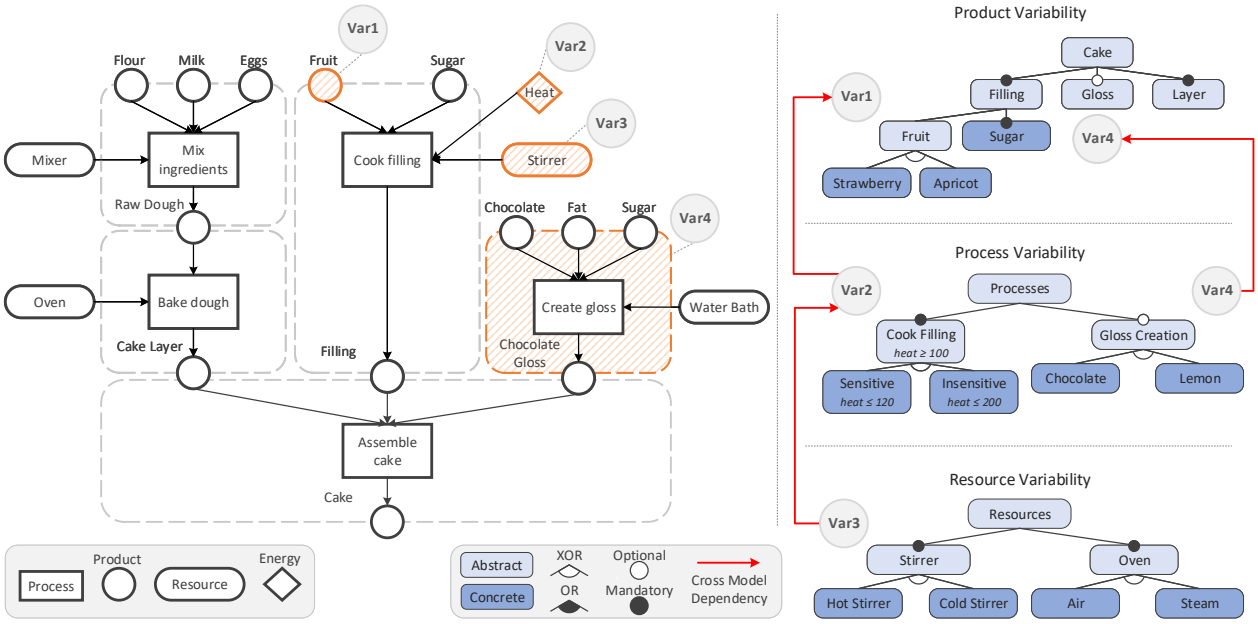


Figure 1: *Cake baking* CPPS example represented by a *PPR* model (left) and orthogonal feature models (right) to describe Products, Processes, and Resources.

be, e.g., the trade-off between dough creation throughput and the disintegration of the *Eggs* due to the heat generated by the *Mixer*.

This simple yet realistic *cake baking* example is an academic abstraction of our industry partner's *CPPSs* that we use to improve common understanding and due to non-disclosure agreements. It still shows the complexity that *CPPSs* can imply on variability modeling. Indeed, there is scientific evidence on the complexity of *cake baking CPPSs* in food engineering [31, 46]. One can easily transfer the example to any *CPPS* in the industrial automation domain, e.g., product aspects could be machine parts to be built, gloss could be varnish; process aspects could be the welding of machine parts (using different welding methods and heat levels) in discrete manufacturing, painting in continuous production; resources could be welding equipment or a painting robot. The cake baking example allows identifying and discussing challenges that variability modeling methods face in a *PPR*/*CPPS* context.

5 CHALLENGES

To answer *RQ1*, we identified the following challenges for *VM* in the context of *PPR* for *CPPSs* based on the described use case and discussions with domain experts of our industry partner.

CH.1 – Multiple Disciplines. In most case study reports [5, 32, 55], the modelers, who actually create variability models, typically either come from the same or similar domains, like software engineering, and/or do this with insufficient tools like *Excel*. In common practice, there is just one person responsible for the variability modeling, if there is any person. However, in *CPPS* engineering, several modelers are coming from very heterogeneous domains, such as mechanical, electrical, process, fluidic, and software engineering, with their discipline-specific tools and vocabularies. These different views imply an additional dimension in the variability models, as each of the domains brings in its own perspective on the particular *PPR* variability aspects. Existing approaches that emphasize multi-disciplinary models [1, 16, 24] could be a good starting point to develop an approach for this context.

To accurately model the variability in such heterogeneous multi-disciplinary environments, we identified three sub-challenges to address. (a) *Common concepts.* Engineers need to agree on a set of common concepts [35] defining variation points in their specific domain to later identify the connecting points representing dependencies between the variability representations. (b) *Variability dependencies.* Engineers then need to find out which changes in a discipline-specific representation need to be propagated to dependent representations to model variability consistently. For instance, safety constraints in connected variability models might be crucial for the correct operation of a *CPPS* and, therefore, are important to be preserved. (c) *Change awareness.* Newly introduced or modified concepts or variation points need to be communicated to the stakeholders of dependent domains to allow them to adapt their variability models accordingly and to ensure consistency. Challenge *CH.1* impacts several of the following identified challenges due to its cross-cutting nature.

CH.2 – Heterogeneity. Krüger et al. [27] affirmed that *CPSs* combine heterogeneous aspects, requiring a combined representation of the systems' variability. The authors identified three sources of heterogeneity in *CPSs*, (a) a broad range of different *artifacts*, (b) various levels of *information granularity*, and (c) mixed *representations of variability*, and proposed using either a mapping between the variability models or an integrated model for the aspects.

The *cake baking* use case demonstrates that the identified sources of heterogeneity similarly exist for *PPR* variability modeling. In our

case, (a) the range of artifacts is the distinct models for the three different aspects of *PPR*, namely the product, process, and resource variability models, (b) the varying levels of information granularity can also be found in the *PPR* case as, e.g., the gloss product might be connected to more than one process step, and (c) the mixed representations that can be mapped to different modeling approaches or hierarchies of the *PPR* aspects.

Looking at Figure 1, one can tell that representations like feature models can help to structure variability but, at the same time, these models and their mutual dependencies may quickly become large, complex, and, subsequently, hard to manage for practitioners. While approaches exist to deal with product variability [11], process variability [52], and resources [51], there is no integrated approach for dealing with products, processes, and resources as well as their variability. A very specific approach that aims at reusing safety cases for safety-critical product development processes by combining these aspects was presented in [17]. For the practical *CPPS* context, it, however, remains unclear whether to prefer an integrated modeling approach or an approach with multiple separate/orthogonal models and explicit mappings.

CH.3 – Usability. Due to the multi-disciplinary environment, stakeholders from various domains will be involved in the variability modeling process. Therefore, the usability of the modeling tools for different types of users from the *CPPS* engineering domain is essential for a broad acceptance, which is hard to achieve when having to deal with user groups with diverse backgrounds. To this end, we also count quality in terms of, e.g., interdisciplinary correctness of the variability models, to the characteristics of usability.

Currently, variability modeling methods and tools are often academic solutions that do not have usability as their primary focus, particularly not in reported evaluations [10]. Those tools that do emphasize usability, e.g., the commercial tools *pure::variants*[2], *Gears*[3], and the (openly available) *Feature IDE*[4], have still mainly been designed envisioning users with a software (engineering) background. While some work has been investigating the usability of product configuration tools [40], to the best of our knowledge, the usability of variability modeling tools has not been considered, especially not in a *CPPS* context.

The general dilemma between automation and usability has been discussed before [37]. The key challenge in our context is to provide concepts and tools that help practitioners to model variability efficiently, even in a heterogeneous context, while maintaining the correctness at a reasonable level of complexity. Furthermore, the engineering tools of the particular domain experts have to employ the provided concepts to foster their usage in practice, while still maintaining engineering domain-specific vocabulary.

CH.4 – (Co-)Evolution. CPPS engineering is driven by requirements of industry and customers as well as by frequently evolving technologies. Therefore, the concepts represented in *PPR* variability models are subject to continuous change. The underlying *PPR* aspects, such as the production processes, are, in principle, evolutionary independent, which means that an engineer typically will change these aspects without notifying others. For instance, the DIN 8580 [14] provides a standardized catalogue for manufacturing processes that evolves independently from production resources existing on the market. However, a change in one aspect and its variability model may require the adaptation of dependent models. For instance, if a new production process is developed, variability may change not only the process variability model but also the resource variability model, if new requirements towards resources are introduced. Similarly, if, e.g., a new sort of fruit for a cake filling has to be considered, new or changed dependencies must be propagated between the product and process variability models. Remarks from engineers revealed that today, such evolutionary changes are done unsystematically in various tools and artifacts, with a propagation among domains often only on demand and, e.g., via e-mail or in informal meetings, causing issues during integration.

Such changes not only have to be considered when supporting the co-evolution of the variability models and their dependencies but also in the means to propagate such changes to participating engineers, which highly relates this challenge to *CH.1*. Existing research on the co-evolution of variability models and product line artifacts [9, 13, 30, 50] is a promising starting point to address this challenge, but, to our best knowledge and according to the reviewed literature, so far has not been applied in the *CPPS* context. Other work on co-evolution, e.g., of products, processes, and production systems in the manufacturing domain [54], has not considered variability systematically. A key goal is to achieve, at some point, an integrated, consistent variability model that can safely be used in the engineering phase. Developing a consistency checking approach [57, 58] to detect and fix inconsistencies between (models for) products, processes, and resources thus could be a promising first step to support co-evolution.

6 TOWARDS PPR VARIABILITY MODELING

To address the challenges described in Section 5, and following other discussions of *CPS* challenges [27] and issues regarding integrating software process and product lines [43], we aim at developing an approach to support integrated *VM* for *CPPS*. Instead of re-inventing the wheel, we build on existing *VM* approaches discussed in Section 2 and adapt and combine these approaches as needed. Promising starting points are *orthogonal VM approaches*, such as *Decision Modeling (DM)* [11] and *OVM* [38]. However, one also has to consider the availability of tools that support modeling products, processes, and resources, and their variability in an integrated manner, while supporting multi-disciplinary users, in a context with high demands on usability and continuous evolution.

6.1 Preliminary Industry Case Study: Assembly Sequence PPR VM for Rocker Switches

Together with domain experts at our industry partner, we conducted a case study on the usefulness and usability of means to model variability by extending the semantics of the *FPD* for *PPR* modeling. Figure 2 shows the *FPD* model extension in the context of a section of an *assembly sequence* of a *rocker switch PPR* model[5] The variants of the switches are based on real-world *CPPS* use cases from our industry partner, elicited from several documents and

[2] pure::variants - https://www.pure-systems.com/
[3] Gears - https://biglever.com/solution/gears/
[4] Feature IDE - http://www.featureide.com/

[5] *Rocker switches* have one or more rockers, e.g., realized as single-pole-single-throw, double-pole changeover, or four-way switches, controlling devices like sun-blinds.

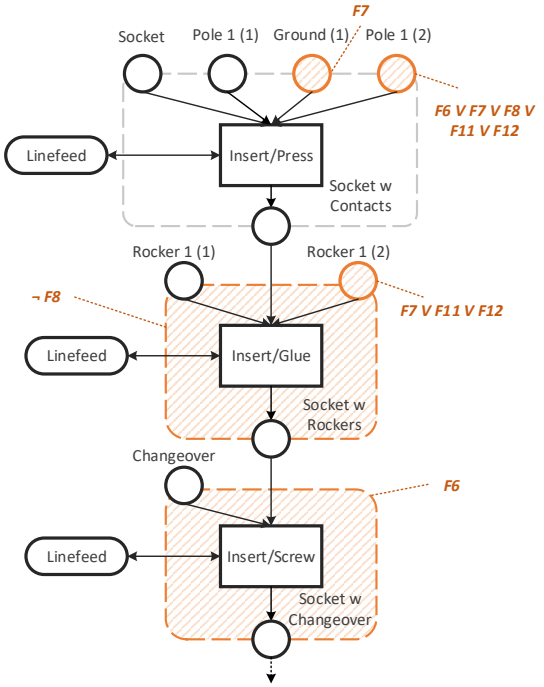


Figure 2: *PPR* model of six *rocker switch assembly sequence* variants related with different features. Numbers in brackets indicate multiple instances of elements.

Excel sheets (~300 rows × 45 columns) that engineers already use as variant matrices for products. Instead of presenting a single *PPR* model variant that engineers subsequently clone, this model represents the union of several variants, following an annotative, 150% modeling approach [47].

In the notation, we depict mandatory *PPR* elements in black, while optional/alternative elements are filled with a hatched pattern. Selection of optional/alternative[6] elements depends on the selection of certain (obfuscated) features annotated in the figure. For example, while in Figure 2 the first process is mandatory for all variants including the *Socket* and *Pole 1 (1)*, *Ground (1)* is only needed for a particular feature (F7).

When a process step is marked with a hatched pattern, all of its input factors and resources are optional. Nested optional elements are further marked with a hatched pattern. For instance, the second (insert/glue) process step in the *assembly sequence* is needed in all variants except for when *Feature 8* was chosen, as shown by the negation in the feature annotation. The second *Rocker 1 (2)* in the second process step is relevant for features *F7*, *F11* and *F12*, indicated by the pattern and the usage of the logical *OR* conjunction. This means when selecting *F7*, the optional elements *Ground (1)*, *Pole 1 (2)*, and the second process step including *Rocker 1 (2)* are needed besides the mandatory *Socket* and the *Pole 1 (1)*.

This extension of the notation addresses challenge *CH.3* as, in practice, domain experts in basic engineering first need to identify the relevant product variants from customer requirements and determine their particular *assembly sequence*. Therefore, domain experts require a *PPR* notation extension that enables them to model *assembly sequence* variants before building up the rest of the variability model. We argue that such models are useful and usable for *CPPS* engineers to express the variants of the *assembly sequence*.

To this end, we investigated in the case study the notation extension for twelve *assembly sequences* of *rocker switch* variants. In an open interview with three domain experts at the industry partner, we discussed the model to evaluate our approach. We explained the *PPR* model notation, in particular, the extensions introduced for modeling variability. The experts provided feedback on the completeness of the model and on model expressiveness regarding relevant variants. The domain experts found the model extension to improve their way of modeling the variants of a product in comparison to their traditional way of describing variants in *Excel* spreadsheets. The domain experts also confirmed that the current version of the approach provides the required means to represent product variants properly in a single model. The experts also found the model useful to communicate their ideas to partner engineers and customer representatives. However, the experts mentioned that they would require specific tool support to design, analyze, and maintain the potentially complex model.

This preliminary case study is a first step towards answering *RQ2* (cf. Section 3) and developing an approach to support integrated variability modelling of *PPR* in a *CPPS* context.

6.2 Status and Research Agenda

While the examples presented in this paper allowed us to initially describe the challenges (cf. Section 5) for *VM* in the context of *PPR* for *CPPS*, to investigate *RQ1* (cf. Section 3) in sufficient detail, we need to systematically study existing *VM* approaches. We already have started conducting a *systematic mapping study* to identify and analyze *VM* approaches in literature that (a) provide at least basic support for integrated modeling of product and process variability, (b) are extensible, and (c) have the potential to be applicable to the domain of *CPPS* to (partly) address the found challenges (cf. Section 5. As there is already a plethora of systematic literature reviews and mapping studies [3, 5, 10, 11, 18], we conduct a tertiary study, similar to the study by Raatikainen et al. [39], but with a different focus: we put emphasis on finding approaches that are good in dealing with multiple disciplines (cf. *CH.1*), heterogeneity (cf. *CH.2*), and (co-)evolution (cf. *CH.4*). Tool support that has been evaluated with real users (cf. *CH.3*) will also be a major criterion.

To fully address *RQ2* (cf. Section 3), we will build on the findings of this study and on our modeling experiments with industry partners to (a) further adapt the *FPD* approach for *Variability Modeling*, e.g., by introducing abstract aspects, and (b) propose an approach for designing an *Integrated Variability Model (IVM)* for *PPR*. The *IVM* should provide a basis to model the relevant information for decision-making, such as the properties of processes and resources, and dependencies between the *PPR* aspects to build up an orthogonal variability model representing variability in the problem space. For example, a decision model could serve as a basis for detail engineering to narrow down suitable resources for a *CPPS*.

To address challenge *CH.1 – Multiple Disciplines* and challenge *CH.3 – Usability*, we plan to employ *Collective Intelligence Systems*

[6]Note that at this point, we do not explicitly separate between optional and alternative elements in the graphical notation.

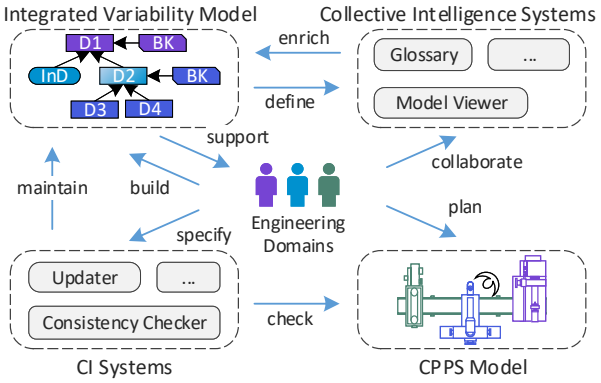


Figure 3: Draft Solution Architecture for Integrated Variability Modelling of PPR in a CPPS context.

(CISs) [36] to negotiate common concepts [35] and allow collaborative work on the *IVM* by providing discipline-specific views [25]. We have already collected engineering terms in a multi-user glossary farm[7] as a first step. To address the correctness of the *IVM* for different disciplines and to support (co-)evolution (cf. *CH.4*), we plan to apply human-supported inspection to the models [7] and adapt automated consistency checking techniques [57, 58].

Figure 3 shows a first architecture draft of our solution approach, with the *IVM* on the upper left, *CISs* examples on the upper right, a *CPPS* model on the lower right, and automated services like *Continuous Integration (CI)* systems on the lower left. Engineers from different domains are supported by the *IVM* when planning the *CPPS*. Therefore, engineering tools used by the different disciplines facilitate the *IVM* to gather relevant information concerning the perspectives of the *CPPS*. For example, when a mechanical engineer wants to decide which robot arm to use as a particular resource, a tool might provide a selection based on the properties of the product or process. The *IVM* is created and maintained by the engineers using diverse collaborative *CISs*, e.g., the glossary, that enrich the *IVM* and provide domain-specific views. Vice versa, the *IVM* defines possible selections for modeling, e.g., in a shared *PPR VM* editor. The *CI* systems check and maintain the consistency of the *IVM* and the *CPPS*, e.g., based on specific rules, and update the *IVM* if required. Such a system could, e.g., be a *CI* server that checks the consistency of the *CPPS* engineering artifacts, like electrical plans, and, in parallel, maintains the consistency of the *IVM*.

After building a first prototypical design, we will conduct further case studies with industry partners to investigate the feasibility and effectiveness of our *PPR* variability approach to address the described challenges. Based on the results of these studies, we will iteratively improve and evaluate the approach in a larger context.

7 CONCLUSION

When planning and engineering *CPPS*, engineers from heterogeneous disciplines have to work together, making effective and efficient knowledge exchange crucial. The *PPR* approach and the *FPD* language provide foundations for knowledge exchange in *CPPS* by representing the products to be manufactured in combination with the required production processes and resources. As *CPPSs* usually manufacture a broad product portfolio with many product variants, engineers also need to model variability in a suitable way.

In this paper, we raised *RQ1*, asking which aspects of *CPPS* mainly challenge existing *VM* approaches. We described a simple yet realistic example of a *cake-baking CPPS* to identify four key challenges for variability modeling in *CPPSs*: *multi-disciplinary views*, *heterogeneity* of artifacts, *usability* of *VM* methods/tools, and *(co-)evolution* of product and production-process variability models. We argue that existing *VM* solutions fall short in addressing (all of) these challenges and that orthogonal approaches are best suited to create a variability model that represents different views and dependencies.

RQ2 asked how the *PPR* modeling notation can be extended as foundation for *(co-)evolution*. Interviews with domain experts revealed that first the challenge of *usability* needs to be addressed, i.e., allowing them to model variants of a basic artifact such as the *assembly sequence* of the products to build in a straightforward and integrated way. An initial evaluation of a *PPR* notation extension in a case study found potential for *PPR* modeling to help engineers describe better model variants and to explain *assembly sequence* variants to other engineers as well as customers. The *PPR* extension should serve as foundation to investigate how to tackle further challenges identified by *RQ1*. Moreover, we proposed a basic solution architecture for *VM* of *PPR* in the context of *CPPS* and a research agenda to address these challenges in the *CPPS*/*PPR* context.

As next step we plan to integrate a first orthogonal variability model describing standard processes used in manufacturing, such as the DIN 8580, to the *PPR* notation extension to further investigate the feasibility of the orthogonal approach in industrial contexts.

ACKNOWLEDGMENTS

The financial support by the Christian Doppler Research Association, the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

REFERENCES

[1] Sofia Ananieva, Matthias Kowal, Thomas Thüm, and Ina Schaefer. 2016. Implicit constraints in partial feature models. In *Proc. of the 7th Int. FOSD Workshop, FOSD@SPLASH 2016, Amsterdam, Netherlands, October 30, 2016.* 18–27.
[2] Sven Apel, Don Batory, Christian Kaestner, and Gunter Saake. 2013. *Feature-Oriented Software Development: Concepts and Implementation.* Springer.
[3] Rabih Bashroush, Muhammad Garba, Rick Rabiser, Iris Groher, and Goetz Botterweck. 2017. CASE Tool Support for Variability Management in Software Product Lines. *Comput. Surveys* 50, 1 (2017), 14:1–14:45.
[4] Luca Berardinelli, Alexandra Mazak, Oliver Alt, Manuel Wimmer, and Gerti Kappel. 2017. Model-driven systems engineering: Principles and application in the CPPS domain. In *Multi-Disciplinary Engineering for Cyber-Physical Production Systems.* Springer, 261–299.
[5] 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 *Proc. of the 7th Int. Workshop on Variability Modelling of Software-intensive Systems.* ACM, 7–14.
[6] Stefan Biffl, Detlef Gerhard, and Arndt Lüder. 2017. Introduction to the Multi-Disciplinary Engineering for Cyber-Physical Production Systems. In *Multi-Disciplinary Engineering for Cyber-Physical Production Systems.* Springer, 1–24.
[7] Stefan Biffl, Marcos Kalinowski, and Dietmar Winkler. 2018. Towards an experiment line on software inspection with human computation. In *Proc. of the 6th Int. Workshop on Conducting Empirical Studies in Industry, CESI@ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018.* 21–24.
[8] BKCASE Editorial Board. 2017. *The Guide to the Systems Engineering Body of Knowledge.* Vol. 1.9.1. The Trustees of the Stevens Institute of Technology.
[9] Goetz Botterweck and Andreas Pleuss. 2014. Evolution of software product lines. In *Evolving Software Systems.* Springer, 265–295.

[7] Glossary - https://glossary-farm.herokuapp.com/glossaries/cdl-sqi/terms

[10] Lianping Chen and Muhammad Ali Babar. 2011. A systematic review of evaluation of variability management approaches in software product lines. *IST* 53, 4 (2011), 344–362.

[11] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. 2012. Cool Features and Tough Decisions : A Comparison of Variability Modeling Approaches. In *Proc. of the 6th Int. Workshop on Variability Modelling of Software-intensive Systems*. ACM, 173–182.

[12] Patricia Derler, Edward A Lee, and Alberto Sangiovanni Vincentelli. 2012. Modeling cyber–physical systems. *Proc. of the IEEE* 100, 1 (2012), 13–28.

[13] Deepak Dhungana, Paul Grünbacher, Rick Rabiser, and Thomas Neumayer. 2010. Structuring the modeling space and supporting evolution in software product line engineering. *JSS* 83, 7 (2010), 1108–1122.

[14] DIN. 2003. DIN 8580 – Manufacturing processes. https://standards.globalspec.com/std/756719/DIN%208580 [Online; accessed 2019-04-02].

[15] Hoda A. ElMaraghy. 2009. Changing and evolving products and systems–models and enablers. In *Changeable and reconfigurable manufacturing systems*. Springer, 25–45.

[16] Stefan Feldmann, Christoph Legat, and Birgit Vogel-Heuser. 2015. Engineering support in the machine manufacturing domain through interdisciplinary product lines: An applicability analysis. *IFAC-PapersOnLine* 28, 3 (2015), 211–218.

[17] Barbara Gallina. 2015. Towards Enabling Reuse in the Context of Safety-critical Product Lines. In *Proc. of the 5th Int. Workshop on Product LinE Approaches in Software Engineering (PLEASE '15)*. IEEE Press, Piscataway, NJ, USA, 15–18.

[18] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. 2014. Variability in Software Systems - A Systematic Literature Review. *IEEE TSE* 40, 3 (mar 2014), 282–306.

[19] Carlo Ghezzi and Amir Molzam Sharifloo. 2013. Model-based verification of quantitative non-functional properties for software product lines. *IST* 55, 3 (2013), 508–524.

[20] Volkan Gunes, Steffen Peter, Tony Givargis, and Frank Vahid. 2014. A survey on concepts, applications, and challenges in cyber-physical systems. *KSII Transactions on Internet & Information Systems* 8, 12 (2014).

[21] Gerald Holl, Paul Grünbacher, and Rick Rabiser. 2012. A Systematic Review and an Expert Survey on Capabilities Supporting Multi Product Lines. *IST* 54, 8 (2012), 828–852.

[22] ISA 95 2003. IEC 62264-1 Enterprise-control system integration–Part 1: Models and terminology. IEC, Genf.

[23] Lukas Kathrein, Arndt Lüder, Kristof Meixner, Dietmar Winkler, and Stefan Biffl. 2019. Production-Aware Analysis of Multi-disciplinary Systems Engineering Processes. In *Proc. of the 21st Int. Conf. on Enterprise Information Systems - Vol. 2: ICEIS,*. INSTICC, SciTePress, 48–60.

[24] Matthias Kowal, Sofia Ananieva, Thomas Thüm, and Ina Schaefer. 2017. Supporting the Development of Interdisciplinary Product Lines in the Manufacturing Domain. *IFAC-PapersOnLine* 50, 1 (2017), 4336–4341.

[25] Max E Kramer, Erik Burger, and Michael Langhammer. 2013. View-centric engineering with synchronized heterogeneous models. In *1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. ACM, 5.

[26] Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. 2016. Comparing Algorithms for Efficient Feature-model Slicing. In *Proc. of the 20th SPLC (SPLC '16)*. ACM, New York, NY, USA, 60–64.

[27] Jacob Krüger, Sebastian Nielebock, Sebastian Krieter, Christian Diedrich, Thomas Leich, Gunter Saake, Sebastian Zug, and Frank Ortmeier. 2017. Beyond Software Product Lines: Variability Modeling in Cyber-Physical Systems. In *Proc. of the 21st SPLC - Vol. A*. ACM, New York, NY, USA, 237–241.

[28] Anna-Lena Lamprecht, Stefan Naujokat, and Ina Schaefer. 2013. Variability Management beyond Feature Models. *Computer* 46, 11 (2013), 48–54.

[29] Edward Lee. 2015. The past, present and future of cyber-physical systems: A focus on models. *Sensors* 15, 3 (2015), 4837–4869.

[30] Sascha Lity, Sophia Nahrendorf, Thomas Thüm, Christoph Seidl, and Ina Schaefer. 2018. 175% Modeling for Product-Line Evolution of Domain Artifacts. In *Proc. of the 12th VAMOS 2018, Madrid, Spain, February 7-9, 2018*, Rafael Capilla, Malte Lochau, and Lidia Fuentes (Eds.). ACM, 27–34.

[31] Mathieu Lostie, Roman Peczalski, and Julien Andrieu. 2004. Lumped model for sponge cake baking during the "rust and crumb"period. *Journal of Food Engineering* 65, 2 (2004), 281–286.

[32] Jabier Martinez, Wesley KG Assunção, and Tewfik Ziadi. 2017. ESPLA: A catalog of Extractive SPL Adoption case studies. In *Proc. of the 21st SPLC-Vol. B*. ACM, 38–41.

[33] László Monostori. 2014. Cyber-physical Production Systems: Roots, Expectations and R&D Challenges. *Procedia CIRP* 17 (2014), 9–13.

[34] Richard Mordinyi and Stefan Biffl. 2015. Versioning in Cyber-physical Production System Engineering: Best-practice and Research Agenda. In *Proc. of the 1st Int. Workshop on Software Engineering for Smart CPS (SEsCPS '15)*. IEEE Press, Piscataway, NJ, USA, 44–47.

[35] Thomas Moser and Stefan Biffl. 2012. Semantic Integration of Software and Systems Engineering Environments. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 1 (Jan. 2012), 38–50.

[36] Juergen Musil, Angelika Musil, Danny Weyns, and Stefan Biffl. 2015. An architecture framework for collective intelligence systems. In *2015 12th Working IEEE/IFIP Conf. on Software Architecture*. IEEE, 21–30.

[37] Andreas Pleuss, Benedikt Hauptmann, Deepak Dhungana, and Goetz Botterweck. 2012. User interface engineering for software product lines: the dilemma between automation and usability. In *Proc. of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM, 25–34.

[38] Klaus Pohl, Günther Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer.

[39] Mikko Raatikainen, Juha Tiihonen, and Tomi Männistö. 2019. Software product lines and variability modeling: A tertiary study. *JSS* 149 (2019), 485–510.

[40] Rick Rabiser, Paul Grünbacher, and Martin Lehofer. 2012. A qualitative study on user guidance capabilities in product configuration tools. In *Proc. of the 27th IEEE/ACM Int. Conf. on Automated Software Engineering*. ACM, 110–119.

[41] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. 2010. Cyber-physical systems: the next computing revolution. In *Proc. of the 47th ACM/IEEE Design Automation Conf.* IEEE, 731–736.

[42] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. 2015. Architecture and behavior modeling of cyber-physical systems with MontiArcAutomaton. *arXiv preprint arXiv:1509.04505* (2015).

[43] Dieter Rombach. 2005. Integrated software process and product lines. In *Software Process Workshop*. Springer, 83–90.

[44] Marcello La Rosa, Wil M P Van Der Aalst, Marlon Dumas, and Fredrik P Milani. 2017. Business Process Variability Modeling: A Survey. *Comput. Surveys* 50, 1 (mar 2017), 2:1–2:45.

[45] Emmanuelle Rouillé, Benoît Combemale, Olivier Barais, David Touzet, and Jean-Marc Jézéquel. 2012. Leveraging CVL to manage variability in software process lines. In *Proc. of the 2012 19th Asia-Pacific Software Engineering Conf.*, Vol. 1. IEEE, 148–157.

[46] Melike Sakin, Figen Kaymak-Ertekin, and Coskan Ilicali. 2007. Simultaneous heat and mass transfer simulation applied to convective oven cup cake baking. *Journal of Food Engineering* 83, 3 (2007), 463–474.

[47] Ina Schaefer, Rick 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. *Software Tools for Technology Transfer* 14, 5 (2012), 477–495.

[48] Miriam Schleipen, Arndt Lüder, Olaf Sauer, Holger Flatt, and Jürgen Jasperneite. 2015. Requirements and concept for Plug-and-Work. *at-Automatisierungstechnik* 63, 10 (2015), 801–820.

[49] Julia Schroeter, Malte Lochau, and Tim Winkelmann. 2012. Multi-perspectives on Feature Models. In *Model Driven Engineering Languages and Systems*, Robert B France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 252–268.

[50] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. 2012. Co-evolution of models and feature mapping in software product lines. In *Proc. of the 16th SPLC-Vol. 1*. ACM, 76–85.

[51] Norbert Siegmund, Marko Rosenmueller, Christian Kästner, Paolo Giarrusso, Sven Apel, and Sergiy Kolesnikov. 2011. Scalable Prediction of Non-functional Properties in Software Product Lines. In *Proc. of the 15th SPLC*. IEEE CS, 160–169.

[52] Jocelyn Simmonds, Daniel Perovich, María Cecilia Bastarrica, and Luis Silvestre. 2015. A megamodel for software process line modeling and evolution. In *Proc. of the 2015 ACM/IEEE 18th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 406–415.

[53] Julio Sincero, Wolfgang Schroder-Preikschat, and Olaf Spinczyk. 2010. Approaching non-functional properties of software product lines: Learning from products. In *Proc. of the 2010 Asia Pacific Software Engineering Conf.* IEEE, 147–155.

[54] T Tolio, Darek Ceglarek, HA ElMaraghy, A Fischer, SJ Hu, L Laperrière, Stephen T Newman, and József Váncza. 2010. SPECIES – Co-evolution of products, processes and production systems. *CIRP annals* 59, 2 (2010), 672–693.

[55] Frank van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering*. Springer Berlin Heidelberg.

[56] VDI/VDE 3682 2005. VDI/VDE 3682: Formalised process descriptions. Beuth Verlag.

[57] Michael Vierhauser, Paul Grünbacher, Alexander Egyed, Rick Rabiser, and Wolfgang Heider. 2010. Flexible and Scalable Consistency Checking on Product Line Variability Models. In *25th IEEE/ACM Int. Conf. on Automated Software Engineering*. ACM, 63–72.

[58] Michael Vierhauser, Paul Grünbacher, Wolfgang Heider, Gerald Holl, and Daniela Lettner. 2012. Applying a Consistency Checking Framework for Heterogeneous Models and Artifacts in Industrial Product Lines. In *Proc. of the 15th Int. ACM/IEEE Conf. on Model Driven Engineering Languages & Systems*. Springer, 531–545.

[59] Birgit Vogel-Heuser and Stefan Biffl. 2016. Cross-discipline modeling and its contribution to automation. *Automatisierungstechnik* 64, 3 (2016), 165–167.

[60] Roland Willmann. 2016. *Ontology matchmaking of product ramp-up knowledge in manufacturing industries : How to transfer a cake-baking recipe between bakeries*. Ph.D. Dissertation. TU WIen.

Towards Efficient Analysis of Variation in Time and Space

Thomas Thüm
TU Braunschweig
Brunswick, Germany

Leopoldo Teixeira
Federal University of Pernambuco
Recife, Brazil

Klaus Schmid
University of Hildesheim
Hildesheim, Germany

Eric Walkingshaw
Oregon State University
Corvallis, USA

Mukelabai Mukelabai
Chalmers | University of Gothenburg
Gothenburg, Sweden

Mahsa Varshosaz
IT University of Copenhagen
Copenhagen, Denmark

Goetz Botterweck
Lero, University of Limerick
Limerick, Ireland

Ina Schaefer
TU Braunschweig
Brunswick, Germany

Timo Kehrer
Humboldt University of Berlin
Berlin, Germany

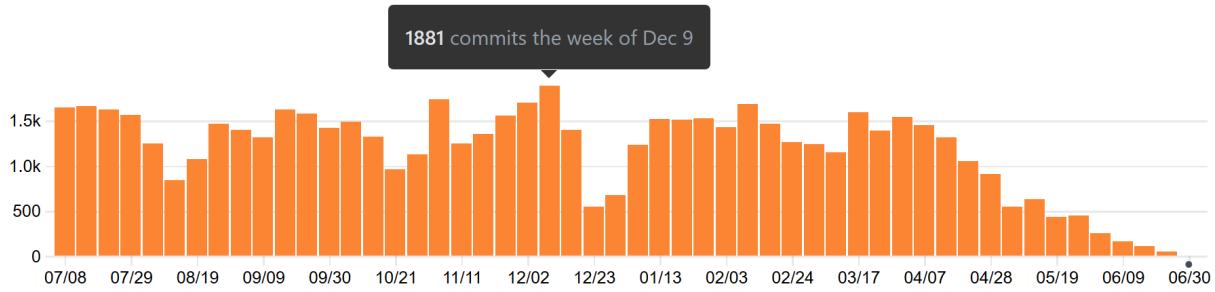


Figure 1: Variation in time: The Linux master branch contains 61,261 revisions (i.e., commits) between July 2018 and June 2019.

ABSTRACT

Variation is central to today's software development. There are two fundamental dimensions to variation: Variation in time refers to the fact that software exists in numerous revisions that typically replace each other (i.e., a newer version supersedes an older one). Variation in space refers to differences among variants that are designed to coexist in parallel. There are numerous analyses to cope with variation in space (i.e., product-line analyses) and others that cope with variation in time (i.e., regression analyses). The goal of this work is to discuss to which extent product-line analyses can be applied to revisions and, conversely, where regression analyses can be applied to variants. In addition, we discuss challenges related to the combination of product-line and regression analyses. The overall goal is to increase the efficiency of analyses by exploiting the inherent commonality between variants and revisions.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software verification and validation**; **Software evolution**.

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342414

KEYWORDS

software configuration management, regression analysis, software product lines, software evolution, software variation, variability management, product-line analysis, variability-aware analysis

ACM Reference Format:
Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards Efficient Analysis of Variation in Time and Space. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3307630.3342414

1 INTRODUCTION

Software development is challenged by two dimensions of variability. The continuous development and improvement of software leads to numerous revisions of the software, which are supposed to replace each other. We refer to revisions as *variability in time*. Due to the high frequency of iterations it is often not feasible to completely analyze each revision and it would involve redundant effort as subsequent revisions are almost identical for large software systems. For instance, the Linux kernel is developed by hundreds of developers giving rise to 61,261 commits within the last year and a peak of 1881 commits in a single week, whereas there are even more revisions which did not make it into the master branch (cf. Figure 1, data accessed on July 1st, 2019). In peak weeks, every five minutes a new revision needs to be analyzed for the master branch (i.e., compiled and tested during continuous integration).

Besides variability in time, software is often developed in different variants that are designed to co-exist simultaneously. We refer to those software variants as *variability in space*. There are many reasons for the development of software variants, such as alternative hardware, conflicting requirements, or the optimization of non-functional properties. While for a low number of variants clone-and-own may be used, many variants are developed with dedicated implementation techniques in a product line [6, 14]. Product lines enable to generate software variants (a.k.a. products) for a selection of features. For instance, Linux has about 18,000 features which are mapped to implementation artifacts by means of the C preprocessor. While the exact number of features depends on the architecture and revision, the number of variants grows exponentially with the number of features [47]. To the best of our knowledge, the exact number of variants is not even known for Linux. However, it is consensus in the product-line community that it is not feasible to analyze them all separately [1, 17, 21, 22, 25, 29, 41, 46].

The massive variation imposed by revisions and variants requires efficient analysis techniques [50]. We recognize that analyses for revisions and variants have been proposed by largely different communities. We refer to analyses that have been designed to efficiently analyze revisions as regression analyses. Examples for regression analyses are regression testing [54], change impact analysis [9, 24], incremental program analysis [7, 13, 45], and regression verification [19, 23]. In contrast, analyses devoted to the analysis of variants are known as product-line analyses [47, 53]. Our goal is to bridge the gap between those communities by systematically discussing how regression analyses can be applied to variants and product-line analyses to revisions. Furthermore, we discuss how variation according to both dimensions, namely time and space, can be efficiently analyzed. We refer to such analyses as *product-line regression analyses*. Figure 2 gives an overview on the systematic behind our discussions and illustrates the structure of this paper. Overall, we make the following contributions:

- We provide a motivating example that illustrates the need for efficient analysis of variants and revisions (Section 2).
- We discuss the application of techniques in both directions, that is, the application of product-line analyses to revisions (Section 3) and the application of regression analyses to variants (Section 4).
- We discuss how product-line analyses and regression analyses can be applied to both dimensions of variation (Section 5).
- Finally, we provide directions for future work that is needed to overcome the identified challenges (Section 6).

2 MOTIVATING EXAMPLE

As a motivating example, we consider a single software product with functionality to store multiple objects in a list. The product p_1 is implemented by a single class called `Store`, as shown in Figure 3. The initial version implements a `read` method that only returns one object—the first object in the list. Later, this implementation is extended by another method to read all objects in the list:

```
public Object[] readAll() { return values; }      (1)
```

This change in the product's implementation constitutes the evolution of the product to a new revision p_1'; we refer to this

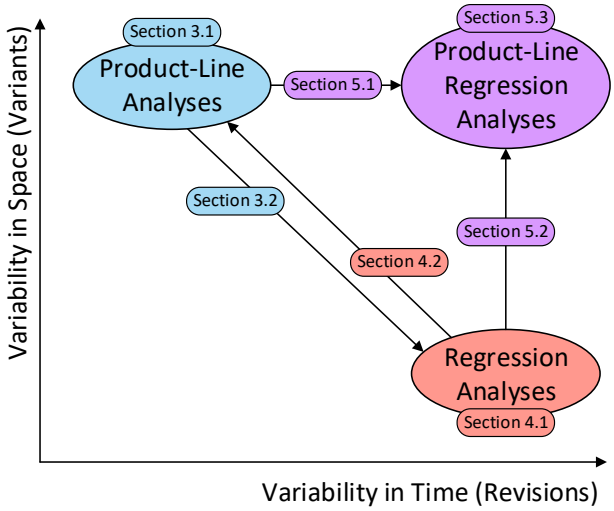


Figure 2: Efficient analyses for two dimensions of variability

evolution as *variation in time*, which is illustrated by the x-axis in Figure 2. Revisions are typically managed by a version control system [12], such as Git or Subversion. In this small example it would be feasible to compile the new revision again from scratch, but for large systems, such as Linux, this compilation can take hours. In the example, we could avoid some checks for the unchanged methods `read` and `set` by means of incremental compilation.

While product p_1' is supposed to replace product p_1, there are also cases where products are intended to co-exist in parallel. In addition to the existing multi-storage functionality, we may need support for single storage (which is less costly) and provide better security to the storage system by introducing access control. We refer to this kind of variation as *variation in space*, as illustrated by the y-axis in Figure 2. To manage variation in space, version control systems often do not scale considering that they are designed to support revisions (variation in time) rather than variants (variation in space). Although branches can be used to handle variation in space to a limited extent [6], the number of required branches can grow exponentially with the number of features due to the combinatorial explosion of possible feature combinations.

With software product-line engineering, product variants are automatically generated for a selection of features [6, 14]. For that

```
1  class Store {
2    private LinkedList values = new LinkedList();
3    Object read() {
4      return values.getFirst();
5    }
6    void set(Object value) {
7      values.addFirst(value);
8    }
9  }
```

Figure 3: A store for multiple objects (product p_1) [47]

```
1  class Store {
2  #IFDEF SingleStore
3    private Object value;
4  #ELSE
5    private LinkedList values =
6      new LinkedList();
7  #ENDIF
8  #IFDEF AccessControl
9    boolean sealed = false;
10 #ENDIF
11   public Object read() {
12 #IFDEF AccessControl
13     if (sealed)
14       throw new RuntimeException(
15         "Access denied!");
16 #ENDIF
17 #IFDEF SingleStore
18     return value;
19 #ELSE
20     return values.getFirst();
21 #ENDIF
22   }
23   public void set(Object value) {
24 #IFDEF AccessControl
25     if (sealed)
26       throw new RuntimeException(
27         "Access denied!");
28 #ENDIF
29 #IFDEF SingleStore
30     this.value = value;
31 #ELSE
32     values.addFirst(value);
33 #ENDIF
34   }
35 #IFDEF MultiStore
36   public Object[] readAll {
37     return values;
38   }
39 #ENDIF
40 }
```

Figure 4: An object store product line implemented with preprocessor annotations.

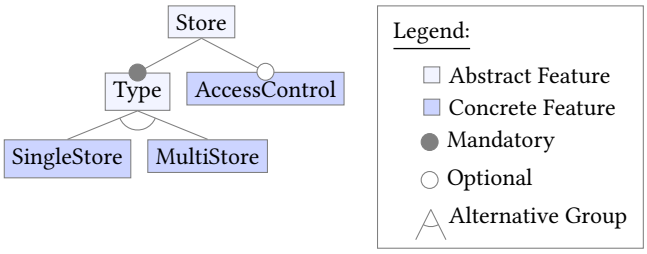


Figure 5: Feature model of the object store product line [47]

to work, valid feature combinations need to be specified and features need to be mapped to code artifacts [6]. The available features and their valid combinations are typically specified by means of feature models as illustrated in Figure 5. Our example has two mandatory features that are mutually exclusive—*SingleStore* and *MultiStore*—any valid product must have one of them, but not both. In addition, we have an optional feature called *AccessControl* that may or may not be included in a product. The feature model specifies a total of four valid configurations each defined by a set of selected features: $c_1 = \{MultiStore\}$, $c_2 = \{SingleStore\}$, $c_3 = \{MultiStore, AccessControl\}$, and $c_4 = \{SingleStore, AccessControl\}$. Figure 4 exemplifies how features can be mapped to code artifacts with conditional compilation. A preprocessor can remove parts of the code prior to compilation based on the selected features. For instance, product p_1' discussed above can be derived automatically by the preprocessor for the configuration c_1.

While our example product portfolio evolves to add new variants in space, variants may also be revised to improve their functionality, resulting in revisions of variants. For instance, the change from product p_1 to p_1' given in (1) may also be applied to an initial revision of the product line and result in adding the lines 35–39 of Figure 4. As the reader may have noticed, we introduced a type error by letting method `readAll` return a list of type `LinkedList` instead of the specified return type `Object[]`. A further revised implementation of the `readAll` method could be:

```
public Object[] readAll() { return values.toArray(); }  (2)
```

This would consequently lead to new revisions of variants derived for the configurations c_1 and c_3 since they contain feature *MultiStore*. Even though the compilation error is resolved with this revision of the product line, unit testing could uncover a security problem with method `readAll`, as it grants access to sealed object stores. Copying lines 12–16 to the beginning of method `readAll` would fix that problem, but results in a further revision of the product line.

The detection of the compilation error and the security problem is not straightforward for a product line. A particular challenge is that we cannot simply compile and test the code without preprocessing. In our example, a compiler would identify unreachable code in Line 20, whereas the lines 18 and 20 are never included in the same product. The brute-force strategy is to run the preprocessor with every possible configuration followed by the actual analyses (e.g., compilation and testing). This strategy would identify the same compiler error in two products and, thus, involve redundant effort. While the brute-force strategy is feasible in our tiny example, it cannot be applied to Linux with up-to $2^{18,000}$ product variants.

In the past two decades, numerous approaches have been proposed to analyze product lines more efficiently than with the brute-force strategy [47]. However, they typically only focus on efficient analysis of variants and not revisions. In contrast, regression analyses focus on revisions, but not variants. In the following, we discuss how to efficiently analyze variation according to a single dimension of variation and with respect to both dimensions of variation.

3 APPLYING PRODUCT-LINE ANALYSES TO VARIATION IN TIME

In this section, we discuss work on product-line analysis, that is, analyses of software systems that explicitly consider variability in space as depicted in Figure 2. Such an analysis exploits knowledge about variability in some way to enable the efficient analysis of product lines as we discuss in Section 3.1.

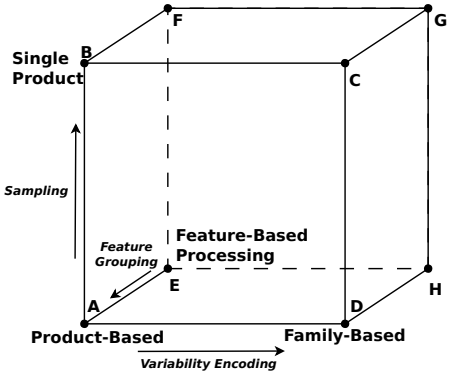


Figure 6: The product-line analysis cube visualizes the space of possible combinations of product-line analyses [53].

In case one wants to analyze different revisions over time in an integrated way, one can regard these different revisions as analogous to variants. This opens the possibility of *applying* product-line analyses to revisions, depicted as an arrow leading from product-line analyses (top left) to regression analyses (bottom right) in Figure 2. We explore this concept further in Section 3.2.

3.1 Product-Line Analyses

The idea of product-line analyses is to establish properties for all products of a product line. Basically any of existing product analysis can be lifted to the product-line level. These can be static analyses like dead-code analyses, which aim at identifying code parts that are not part of any variant [46] or type checking of product lines that aims at identifying whether any variant violates typing rules [11, 27]. For instance, type checking can find the compilation error for configuration c_1 and c_3 discussed in Section 2.

Many different analyses have been lifted to the product line level [47]. However, the basic strategies applied can be divided into three different categories: product-based, feature-based, and family-based analysis as well as combinations thereof [47]. A *product-based analysis* aims at analyzing the property for each product, individually, concluding the property when all products have been shown to exhibit the property. *Feature-based analysis*, in a similar way, analyses the property for each feature in the product line. Finally, *family-based analysis* aims to perform the analysis for the whole range of domain assets in an integrated way, taking the relevant variability explicitly into account.

The product-line analysis cube shown in Figure 6 builds on those basic strategies and extends it to a formalization of product-line analysis strategies [53]. It introduces three different strategies for variability analysis: sampling, feature grouping, and variability encoding. *Sampling* refers to the strategy of selecting a subset of products and performing a product-level analysis on them. Using an adequate sampling heuristic, this aims to establish a high probability that if the property under analysis holds for the subset, it will hold for all products. For instance, it would be sufficient to have either c_1 or c_3 in the sample to detect the type error of our running example. *Feature grouping* refers to the concept of focusing on adequately selected subsets of features. Feature grouping does not necessarily say anything regarding the completeness of the features to be addressed. It can be combined with sampling, that is, only addressing some features in total and making a heuristic argument regarding the completeness, or it can select a subset of features that allow for the sound conclusion that the property holds for all features, if it can be shown for the subset [43]. It can also be a way to subdivide the range of features, but all will be taken into account over the complete analysis. Finally, *variability encoding* describes the strategy of explicitly encoding the variability and using this information within the analysis. The product-line analysis cube allows designers of product-line analyses to choose any point in the cube and to build such an analysis, whereas it depends on the product line and type of analysis which point in the cube leads to the most efficient analysis [53].

3.2 Reusing Product-Line Analyses for Revisions

In principle, we can interpret a set of revisions as a set of products in the sense of a product line. Thus, we can interpret variation in time as variation in space, providing a pathway to apply product-line analysis also to revisions. This way, product-line research and tools may be applied as-is for the analysis of variation in time.

We can regard the differences that are created by updates from one revision to another as features that exist in the product line. For instance, the two revisions of the *MultiStore*, presented in (1) and (2) in section 2, could be threated as two separate features—one that reads only one value from the store and another that can read all values. This would allow for the application of feature-based analysis approaches to revisions. To the best of our knowledge, this has not yet been done, but it would seem to be straightforward with delta-based approaches like delta-oriented product testing [3].

A different angle would be to use a sufficiently expressive variation representation, which allows encoding revisions over time in the same way that we represent variation in space. For example, with an annotative variation representation such as #ifdefs (see Listing 4) or the choice calculus [18], we can introduce a "feature" corresponding to each revision of the program. Similarly, in delta-modeling, revisions can be captured by deltas [31]. By encoding revisions as variants, we can directly apply family-based analyses designed for software product lines to entire revision histories. The ability to efficiently analyze a large number of revisions in time as well as in space opens up new potential applications, such as determining when a particular behavior or interaction was introduced, supporting sophisticated ways of selectively undoing revisions or cherry-picking patches, and more.

However, product-line analyses have so far hardly been applied to the problem of analyzing revisions at scale. Reasons may be that revision histories can be long and family-based analyses are more expensive than the analysis of individual products. Since we usually only care about a limited number of revisions in the history (e.g., major releases or specific time slices), more general approaches to incrementalize and/or constrain family-based analyses are needed to realize the full potential of such approaches. This opens a new field for further research.

4 APPLYING REGRESSION ANALYSES TO VARIATION IN SPACE

In this section, we discuss work related to analyses applied in the context of variability in time, namely *revisions*, as illustrated in Figure 2. We use the term regression analyses to denote works that apply verification and validation techniques over revisions, such as regression testing [54], incremental program analysis [7, 45], and change impact analysis [9, 24]. When considering revisions, the main goal of these techniques is to ensure that the existing software still behaves as expected after a change. We discuss existing approaches which are unaware of variability in Section 4.1. We then present some works that apply regression analyses to the variability context in Section 4.2. This corresponds to the arrow from regression analyses (bottom right) to product-line analyses (top left) in Figure 2.

4.1 Regression Analyses

In theory, to apply regression testing, an approach would be to execute the entire test suite of a system again, to check behavior preservation. However, the number of tests might grow together with a system, making such an approach unfeasible in practice. Therefore, such techniques usually consider what has been changed in a particular evolution scenario, to increase efficiency by defining which tests should be executed, identifying redundant tests, and establishing the priority for ordering test case execution [54]. In our running example, prioritizing a test case containing a call to the *readAll()* function can lead to discovering the fault in the implementation of the function.

A number of works have been proposed targeting incremental analysis, such as separate compilation [13], to avoid recompiling an entire program when there is some change in an interface. Recent works also target incremental program analysis [7, 45], proposing ways for tackling program changes in an efficient way. For instance, *Reviser* [7] extends an existing framework for data-flow analysis to enable recomputing analysis information after a program changes. Its basic intuition is to identify changes based on the control-flow graph of two program versions, and by establishing the origin of changes, proceed on propagating and updating the results of the analysis. *IncA* is a domain-specific language for specifying incremental program analysis, which works in a declarative way over AST representations of programs [45]. For both works, a speedup is observed when compared with running the entire analysis again, while no substantial overhead is introduced. Nonetheless, variability is not considered.

4.2 Reusing Regression Analyses for Variants

If we want to apply regression analyses in the context of variants, one option is to consider variants as revisions, then proceed to apply standard regression analysis. The first issue that arises is the number of variants, which might be challenging to deal with since there might be a huge configuration space. One option is to focus on a subset of the variants to reduce the effort of the analysis. In some cases, it is feasible to consider only those products of interest to particular customers—a common industrial practice [36], and in other cases dedicated sampling techniques are required to derive a sample with certain coverage guarantees [2, 52].

Lity et al. apply regression testing techniques to verify that changes between variants are intended, to systematically exploit reusability of test artifacts [32]. They use delta-oriented models to capture commonalities and variabilities among variants. This approach to modeling is used for reasoning about changes between variants and test artifacts. Testing effort is reduced while maintaining the same degree of test coverage. Heider et al. propose Variability Modeling Regression Testing (VaMoRT) [24]. The approach relies on decision models as variability models, and allows identifying the impact of changes to the variability models on existing products. It avoids testing all variants but rather focuses on the already derived products, since the decision model stores configuration decisions. Based on that, the approach regenerates the products and computes the differences.

Another issue is the actual order in which variants are analyzed. When we are considering variants, change is inherent among them. Therefore, it could be the case that some changes are analyzed more than necessary. Hence, an adequate order in which variants are analyzed may result in minimal changes and avoid rework. There is some work on prioritizing product orders for testing [3, 30]. Al-Hajjaji et al. [3] focus on obtaining higher coverage with the goal of early fault detection. Therefore, it privileges selecting dissimilar products when establishing an order. In contrast, the work by Lity et al. [30] focuses specifically on leveraging an order that might benefit incremental analysis. Thus, it prefers an order in which differences between adjacent products are minimized. Changes among products are captured in regression deltas, and this enables graph algorithms to find an optimal product order. However, this work mainly focuses on comparing products from the same product-line revision. If we extend it to consider revisions of variants, depending on the regression analysis and whether it involves hardware to be reconfigured it could be necessary to have a linear order of variants to be analyzed. This is opposed to a tree-like regression analysis, in which the best match is found for each variant in each of the revisions.

5 EFFICIENT ANALYSES FOR VARIATION IN TIME AND SPACE

In the previous two sections, we discussed how to efficiently analyze the variability induced by either variants or revisions. However, software systems that exist in variants, evolve as well. Thus, as illustrated by the motivating example in Section 2, there is potential and a need for incremental analyses along both dimensions of variation (i.e., time *and* space) simultaneously.

In this section, we aim to discuss different ways in which product-line analyses and regression analyses can be applied to both variants and revisions at the same time. In total, we discuss three different strategies in the following three subsections: in Section 5.1 we discuss how product-line analyses can be homogeneously applied to revisions of variants. In Section 5.2, we complement this by a discussion of how regression analyses can be simultaneously applied to variants and revisions of variants. Finally, in Section 5.3, we discuss the combination of product-line analyses with regression analyses. This structure is also illustrated in Figure 2.

5.1 Applying Product-Line Analyses to Revisions of Variants

While there has been a considerable amount of work on product-line analyses [47], most of it does not aim to save analysis effort when applying it to multiple revisions of a product line. The naïve way of applying any product-line analysis to multiple revisions is to simply analyze every revision from scratch. However, this does not take advantage of typically rather local and comparably small changes to a product line [28]. For instance, in our motivating example, a very small change was made to the implementation of the *MultiStore* feature, affecting only two of the four possible products of the product line; thus, an analysis of the whole product line is unnecessary. Nevertheless, this naïve strategy may serve as a baseline when evaluating the efficiency of more advanced analyses.

One strategy to avoid redundant effort under evolution is to identify which products are affected by the change. A common solution is to classify changes to the product line into refactorings, specializations, generalizations, and arbitrary edits [5, 10, 15, 48]. When applying a refactoring to a product line (i.e., the set of derivable products is identical in both revisions) [49], there is no need to analyze the product line again, as long as it is verified that the change is indeed a refactoring. When applying a specialization (i.e., some products are removed but no new products are added) [15], then an analysis is only necessary if the previous revision did produce errors, as those may be fixed by means of the specialization. For generalizations and arbitrary edits (i.e., changes that add new products), product-line analyses need to run again usually. Borba et al. generalized refactorings, generalizations, and specializations with safe evolution [8], which also incorporate changes to implementation artifacts and not only feature models. Schulze et al. and Pietsch et al. propose a catalog of refactorings [44] and a refactoring construction kit [39, 40] for delta-oriented product line implementations. Sampaio et al.'s extension to partially safe evolution even allows to classify which products are affected by a change [42]. Tool support for classifying changes of the evolution history has been proposed by Dintzner et al. [16].

As product lines can be arbitrarily complex and changes often only have local effects, it has been proposed to modularize product lines into multiple product lines [26, 43, 51]. The overall idea is that a tool checks whether a change is local to an individual product line; if it is indeed local, then it is sufficient to analyze changes only locally and avoid the analysis of all product lines. This imposes some constraints on how the product line is implemented and modeled, but may significantly reduce the effort when analyzing new revisions of the product lines.

Product-line analyses can be organized according to whether their basic unit of analysis is a single product, a feature implementation (e.g., a component or plug-in), or the entire family of products (see Section 3.1). Of these, perhaps *feature-based analyses* best support product-line evolution, because they are inherently incremental. After a revision, one needs only to re-analyze the features that have changed. One challenge is that feature interactions cannot be detected by a feature-based analysis, which is why it is typically combined with a product-based or family-based analysis [47]. So changing a feature implementation may also require re-analyzing other unchanged features that potentially interact with the changed feature. There is likely potential for reuse in these analyses since presumably most code in other features does not interact with the changed feature.

5.2 Applying Regression Analyses to Revisions of Variants

Regression analyses are typically only applied *either* to revisions [7, 9, 24, 45, 54] *or* to variants [4, 47]. Applying a regression analysis to both dimensions of variation is feasible, but there seem to be many opportunities to make it more efficient that are not yet well researched and discussed in the following.

A trivial way to apply regression analysis to revisions of a product line is to follow the strategy described in Section 4.2 to apply it to all variants of a revision and then to apply it from scratch to the next revision, to which we refer to as *variant-only regression strategy*. However, in this case we do not exploit the similarities among the product-line revisions. In contrast, we may exploit the similarities among the product-line revisions, by applying the regression analysis to each variant with respect to its prior revision, to which we refer to as *revision-only regression strategy*. However, then we cannot exploit the similarities among variants within one revision of the product line. Furthermore, it is necessary to have a mapping from variants in the old product-line revision to the variants in the new product-line revision [37]. Such a mapping may not be easy to find depending on the kind of evolution that happened and the implementation technique for variation in space. For delta-oriented product lines, the revision-only regression strategy has been applied in the context of model-based testing [34]. A more advanced strategy, as mentioned in Section 4.2, would be to find the variant with the smallest number of changes to all those that have been analyzed previously. This could include not only analyzed variants of the old product-line revision, but also already analyzed variants from the new product-line version.

As discussed in Section 4.2, the application of regression analyses to product lines typically requires to sample products. When the product line evolves, we can sample the product line again and then apply regression analyses to those sample products. The creation of a new sample can be avoided if none of the artifacts being used as input to the sampling is affected [38]. For instance, if only the feature model is used as input to the sampling algorithm, then we only need to apply the sampling algorithm again on changes to the feature model. However, there are many cases when the computation of a new sample cannot be avoided. While there are many sampling algorithms for product lines [2, 52], they are typically oblivious to the evolution of the product line [38]. In the context of regression analysis, it would be favorable to retrieve a sample for a new revision of the product line that is largely similar to the old sample [37]. The reason is that the efficiency of regression analyses typically depends on the number of changes and similar samples could reduce the number of changes. Besides that, it could be more efficient to adapt the old sample than to start over the sampling algorithm from scratch [37].

5.3 Combinations of Product-Line Analyses and Regression Analyses

So far, we discussed how existing product-line analyses and regression analyses can be reused or extended to support variation in time and space. Besides that, we could combine existing product-line analyses and regression analyses or even invent new product-line regression analyses. Both regression analyses and product-line analyses have in common that they lift an existing analysis to some form of variation, whether variation in time or variation in space. Ideally, we would accomplish an efficient *two-dimensional lifting* by lifting an existing analysis first to variation in time and then to variation in space, or vice versa. Midtgaard et al. proposed to automatically lift existing analyses for variation in space [35], which could be a starting point for the automatic derivation of product-line regression analyses. Two-dimensional lifting has great potential, but it is unclear how much automation is feasible and how efficient automatically derived analyses are.

A direct approach to perform such lifting would be to directly combine an existing product-line analysis with a regression component. As the analysis of real-world product lines shows that individual commits in an evolution history typically have only a very limited impact on the variability in a product line [28], one can expect regression-based extensions of product-line analysis to be very efficient as the updating of analysis results may be a rather local affair. This idea has been applied to dead code analysis over a significant part of the Linux kernel evolution history [20], leading to a speedup over the baseline by an order of magnitude despite the fact that any variability-relevant change of the build model and variability model lead to a complete reanalysis.

Another perspective on product-line regression analyses is to consider time as a fourth dimension in the product-line analysis cube [53]. As discussed in Section 3.1 and illustrated in Figure 6, the existing cube focuses on the mix and match of different analysis strategies with the goal to provide efficient analyses. However, all three existing dimensions, namely variability encoding, sampling, and feature grouping, are mainly focused on variation in space. It is an open research question how to extend the product-line analysis cube to variation in time, but time could be considered as a fourth dimension, or there could be several dimensions for variation in time as there are also three for variation in space.

Another open research question is to what extent the realization techniques for variation in space and variation in time have an impact on the efficiency of product-line regression analyses. For instance, are plug-ins more amendable to analysis than preprocessor annotations? Do we need the change operations that have been applied or is a diff equally good for analysis? Furthermore, it is an open question whether analysis efficiency can be improved if variation in space and variation in time are expressed by the same means, as with higher-order deltas [31] or 175% modeling [33].

6 CONCLUSION AND FUTURE WORK

Today's software development needs to cope with variation in time, variation in space, or even in time *and* space. Crucial for quality assurance and efficient analysis methods that take advantage of commonalities arising from both dimensions of variation. Traditionally, regression analyses are used for variation in time and product-line analyses are used for variation in space.

We discussed fundamental strategies to apply regression analyses and product-line analyses to both dimensions of variation. In particular, we identified how product-line analyses can be applied to analyze variation in time, which seems to be a new application area for many existing product-line analyses. That is, research results of the product-line community could be reused by communities working on regression analyses. While regression analyses have often been applied to variation in space, we summarized common challenges for their application to variants.

With product-line regression analysis, we denote analyses that cope with variation in both dimension, namely time and space. For that purpose, two-dimensional lifting of traditional analyses is necessary with respect to both dimensions of variation. It requires a community effort to identify how to automate the lifting according to both dimensions and which strategies for lifting lead to the most efficient analyses, perhaps also paving the way for an integration of multiple strategies.

ACKNOWLEDGMENTS

In this paper, we summarize the insights of our discussions during a breakout group at Dagstuhl 19191 on *Software Evolution in Time and Space: Unifying Version and Variability Management*. We gratefully acknowledge discussions on sampling for product-line evolution with Sascha Lity, Tobias Pett, Malte Lochau, Sebastian Krieter, and Tobias Runge. This work has been partially supported by the German Research Foundation within the project VariantSync (TH 2387/1-1 and KE 2267/1-1), Science Foundation Ireland (13/RC/2094), FACEPE (APQ-0570-1.03/14), CNPq (409335/2016-9), and the ITEA3-project REVaMP^2, funded by the BMBF (German Ministry of Research and Education) under grant 01IS16042H.

REFERENCES

[1] Iago Abal, Jean Melo, Stefan Stănciulescu, 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] Mustafa Al-Hajjaji, Sascha Lity, Remo Lachmann, Thomas Thüm, Ina Schaefer, and Gunter Saake. 2017. Delta-Oriented Product Prioritization for Similarity-Based Product-Line Testing. In *Proc. Int'l Workshop on Variability and Complexity in Software Design (VACE)*. IEEE, 34–40.

[4] Nauman bin Ali, Emelie Engström, Masoumeh Taromirad, Mohammad Reza Mousavi, Nasir Mehmood Minhas, Daniel Helgesson, Sebastian Kunze, and Mahsa Varshosaz. 2019. On the Search for Industry-Relevant Regression Testing Research. *Empirical Software Engineering* (12 Feb 2019). https://doi.org/10.1007/s10664-018-9670-1

[5] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos José Pereira de Lucena. 2006. Refactoring Product Lines. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 201–210.

[6] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[7] Steven Arzt and Eric Bodden. 2014. Reviser: Efficiently Updating IDE-/IFDS-based Data-flow Analyses in Response to Incremental Program Changes. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 288–298. https://doi.org/10.1145/2568225.2568243

[8] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. 2012. A Theory of Software Product Line Refinement. *Theoretical Computer Science* 455, 0 (2012), 2–30.

[9] Larissa Braz, Rohit Gheyi, Melina Mongiovi, Márcio Ribeiro, Flávio Medeiros, Leopoldo Teixeira, and Sabrina Souto. 2018. A Change-Aware Per-File Analysis

to Compile Configurable Systems with #ifdefs. *Computer Languages, Systems & Structures* 54 (2018), 427–450. https://doi.org/10.1016/j.cl.2018.01.002

[10] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. 2015. Reasoning about Product-Line Evolution Using Complex Feature Model Differences. *Automated Software Engineering* 23, 4 (2015), 687–733.

[11] Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2014. Extending Type Inference to Variational Programs. *ACM Trans. Programming Languages and Systems (TOPLAS)* 36, 1, Article 1 (2014), 1:1–1:54 pages.

[12] Reidar Conradi and Bernhard Westfechtel. 1998. Version Models for Software Configuration Management. *Comput. Surveys* 30, 2 (1998), 232–282.

[13] Regis Crelier. 1994. *Separate Compilation and Module Extension.* Ph.D. Dissertation. Institute for Computer Systems, ETH Zurich. ftp://ftp.inf.ethz.ch/doc/diss/th10650.ps.gz

[14] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* ACM/Addison-Wesley.

[15] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Staged Configuration through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice* 10, 2 (2005), 143–169.

[16] 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.

[17] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2017. An Empirical Study of Configuration Mismatches in Linux. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 19–28.

[18] Martin Erwig and Eric Walkingshaw. 2011. The Choice Calculus: A Representation for Software Variation. *Trans. Software Engineering and Methodology (TOSEM)* 21, 1, Article 6 (2011), 6:1–6:27 pages.

[19] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating Regression Verification. In *Proc. Int'l Conf. on Automated Software Engineering (ASE).* ACM, New York, NY, USA, 349–360. https://doi.org/10.1145/2642937.2642987

[20] Moritz Flöter. 2018. Prototypical Realization and Validation of an Incremental Software Product Line Analysis Approach.

[21] Paul Gazzillo. 2017. Kmax: Finding All Configurations of Kbuild Makefiles Statically. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE).* ACM, 279–290.

[22] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI).* ACM, 323–334.

[23] Benny Godlin and Ofer Strichman. 2009. Regression Verification. In *Proceedings of the 46th Annual Design Automation Conference (DAC '09).* ACM, New York, NY, USA, 466–471. https://doi.org/10.1145/1629911.1630034

[24] Wolfgang Heider, Rick Rabiser, Paul Grünbacher, and Daniela Lettner. 2012. Using Regression Testing to Analyze the Impact of Changes to Variability Models on Products. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 196–205.

[25] 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 *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).* ACM, 805–824.

[26] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A Variability-Aware Module System. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).* ACM, 773–792.

[27] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD).* ACM, 25–32.

[28] Christian Kröher, Lea Gerling, and Klaus Schmid. 2018. Identifying the Intensity of Variability Changes in Software Product Line Evolution. In *Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 1 (SPLC '18).* ACM, New York, NY, USA, 54–64. https://doi.org/10.1145/3233027.3233032

[29] 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.

[30] Sascha Lity, Mustafa Al-Hajjaji, Thomas Thüm, and Ina Schaefer. 2017. Optimizing Product Orders Using Graph Algorithms for Improving Incremental Product-line Analysis. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 60–67.

[31] Sascha Lity, Matthias Kowal, and Ina Schaefer. 2016. Higher-Order Delta Modeling for Software Product Line Evolution. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development (FOSD 2016).* ACM, New York, NY, USA, 39–48. https://doi.org/10.1145/3001867.3001872

[32] Sascha Lity, Malte Lochau, Ina Schaefer, and Ursula Goltz. 2012. Delta-Oriented Model-based SPL Regression Testing. In *Proceedings of the Third International Workshop on Product LinE Approaches in Software Engineering (PLEASE '12).* IEEE Press, Piscataway, NJ, USA, 53–56. http://dl.acm.org/citation.cfm?id=2666064.2666078

[33] Sascha Lity, Sophia Nahrendorf, Thomas Thüm, Christoph Seidl, and Ina Schaefer. 2018. 175% Modeling for Product-Line Evolution of Domain Artifacts. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 27–34.

[34] Sascha Lity, Manuel Nieke, Thomas Thüm, and Ina Schaefer. 2019. Retest Test Selection for Product-Line Regression Testing of Variants and Versions of Variants. *J. Systems and Software (JSS)* 147 (2019), 46–63.

[35] Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski. 2015. Systematic Derivation of Correct Variability-Aware Program Analyses. *Sci. Comput. Program.* 105, C (July 2015), 145–170. https://doi.org/10.1016/j.scico.2015.04.005

[36] 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 *Proc. Int'l Conf. on Automated Software Engineering (ASE).* ACM, 155–166. https://doi.org/10.1145/3238147.3238201

[37] Tobias Pett. 2018. *Stability of Product Sampling under Product-Line Evolution.* Master's thesis. TU Braunschweig, Germany.

[38] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM. To appear.

[39] 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.* IEEE, 852–857.

[40] Christopher Pietsch, Udo Kelter, Timo Kehrer, and Christoph Seidl. 2019. Formal Foundations for Analyzing and Refactoring Delta-Oriented Model-Based Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM. To appear.

[41] Hendrik Post and Carsten Sinz. 2008. Configuration Lifting: Verification Meets Software Configuration. In *Proc. Int'l Conf. on Automated Software Engineering (ASE).* IEEE, 347–350.

[42] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. 2016. Partially Safe Evolution of Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 124–133.

[43] 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.

[44] 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.

[45] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: A DSL for the Definition of Incremental Program Analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016).* ACM, New York, NY, USA, 320–331. https://doi.org/10.1145/2970276.2970298

[46] 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.

[47] 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.

[48] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning about Edits to Feature Models. In *Proc. Int'l Conf. on Software Engineering (ICSE).* IEEE, 254–264.

[49] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract Features in Feature Modeling. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* IEEE, 191–200.

[50] Thomas Thüm, André van Hoorn, Sven Apel, Johannes Bürdek, Sinem Getir, Robert Heinrich, Reiner Jung, Matthias Kowal, Malte Lochau, Ina Schaefer, and Jürgen Walter. 2019. Performance Analysis Strategies for Software Variants and Versions. In *Design for Future—Managed Software Evolution.* Springer, Berlin, Heidelberg. To appear.

[51] Thomas Thüm, Tim Winkelmann, Reimar Schröter, Martin Hentschel, and Stefan Krüger. 2016. Variability Hiding in Contracts for Dependent Software Product Lines. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 97–104.

[52] 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.

[53] Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm, and Ina Schaefer. 2013. The PLA Model: On the Combination of Product-Line Analyses. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 14:1–14:8.

[54] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification and Reliability (STVR)* 22, 2 (2012), 67–120.

Analyzing Variability in 25 Years of Industrial Legacy Software: An Experience Report

Kamil Rosiak
TU Braunschweig
Braunschweig, Germany
k.rosiak@tu-bs.de

Oliver Urbaniak
TU Braunschweig
Braunschweig, Germany
o.urbaniak@tu-bs.de

Alexander Schlie
TU Braunschweig
Braunschweig, Germany
a.schlie@tu-bs.de

Christoph Seidl
TU Braunschweig
Braunschweig, Germany
c.seidl@tu-bs.de

Ina Schaefer
TU Braunschweig
Braunschweig, Germany
i.schaefer@tu-bs.de

ABSTRACT

In certain domains, safety-critical software systems may remain operational for decades. To comply with changing requirements, new system variants are commonly created by copying and modifying existing ones. Typically denoted *clone-and-own*, software quality and overall maintainability are adversely affected in the long-run. With safety being pivotal, a fault in one variant may require the entire portfolio to be assessed. Thus, engineers need to maintain legacy systems dating back decades, implemented in programming languages such as Pascal. *Software product lines (SPLs)* can be a remedy but migrating legacy systems requires their prior analysis and comparison. For industrial software systems, this remains a challenge.

In this paper, we introduce a comparison procedure and customizable metrics to allow for a fine-grained comparison of Pascal modules to the level of individual expressions. By that, we identify common parts of while also capturing different parts between modules as a basis for a transition towards anSPLs practice. Moreover, we demonstrate the feasibility of our approach using a case study with seven Pascal modules totaling 13,271 lines of code with an evolution-history of 25 years and show our procedure to be fast and precise. Furthermore, we elaborate on the case study and detail peculiarities of the Pascal modules, which are characteristic for an evolution-history of a quarter century.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software reverse engineering**; **Software evolution**; *Maintaining software.*

KEYWORDS

Software Prodct Line, Legacy Software, Variability, Clone-and-Own

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342410

ACM Reference Format:
Kamil Rosiak, Oliver Urbaniak, Alexander Schlie, Christoph Seidl, and Ina Schaefer. 2019. Analyzing Variability in 25 Years of Industrial Legacy Software: An Experience Report. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3307630.3342410

1 INTRODUCTION

In certain industrial domains, imperative programming languages such as Pascal are used to implement safety-critical software system functionality. Recent work identified software systems in domains to remain operational for decades [5]. Given such timespan, for engineers to foresee the entire scope of functionality upfront is impossible [25]. Customer requirements may change, regulatory guidelines may be altered, and new business opportunities may emerge. An overall development of complex functionality remains a challenging and time-intensive task. When facing new requirements, engineers frequently resort to an ad-hoc reuse strategy [19]. In *clone-and-own* [22], existing software systems are copied and subsequently modified to create new system variants. This straightforward approach saves time and effort in the short-term, especially when only minor rectification is needed to comply with new requirements [4]. However, proper documentation is typically not applied during *clone-and-own* and relations between individual variants are rendered incomprehensible [3, 24]. With a proliferation of redundant and almost-alike software modules, overall maintainability of the emerging system portfolio is adversely affected [4, 14, 31]. For instance, faults may be unintentionally transferred to a new variant during *clone-and-own*, which can accumulate and account for inexplicable malfunctions at a later time [26, 27]. Consequently, a fault within a Pascal variant may require engineers to assess the entire portfolio to evaluate whether other variants also contain the fault. System rectification then induces a large manual workload and may defer resources from ongoing development, thereby harming portfolio evolution. However, given the nature and application of Pascal modules used in safety-critical environments, their rectification is mandatory. To reinstate sustainable development, SPLs [20] can be a solution [12]. Collapsing redundant parts and capturing relations between similar elements, strategic reuse is facilitated,and maintainability is supported [16, 18]. However, migrating a portfolio of legacy Pascal modules towards an SPL fashion remains an open challenge and requires their prior analysis to identify redundant

and variable parts [32]. This task is further complicated given the modules' evolution history, which can span decades and, therefore, exhibit peculiarities which need to be considered during analysis. In this paper, we try to address this issue and make the following contributions.

- We introduce a model-driven comparison approach for Pascal modules and a customizable metric, which allows for a comprehensive comparison of two systems to the extent of individual expressions.
- We create a procedure to generate a family-model to represent identified variability of compared Pascal modules.
- We demonstrate the feasibility of our approach using an industrial case study, comprising seven Pascal modules with a total of 13,271 lines of code and an evolution history of 25 years.
- We report on our experiences with the case study and discuss identified peculiarities regarding an evolution history of several decades.

The remainder of this paper is structured as follows: We provide background on Pascal software modules, and family-mining in Sec. 2. We introduce our meta-model for Pascal, our customizable metric and our comparison procedure to compare Pascal modules in Sec. 3. We give information on the evaluated Pascal modules and state our methodology in Sec. 4. We discuss produced results and elaborate on identified peculiarities for the case study in Sec. 5. Related work starts in Sec. 6. We conclude our contribution and detail future work in Sec. 7.

2 BACKGROUND

In this section, we state necessary information on Pascal programs and specific properties we utilize for our approach. We briefly elaborate on model-driven development and the concept of family mining.

2.1 Pascal Software Systems

Pascal is an imperative and procedural programming language utilized to realize high-integrity systems [1, 23]. Initially proposed in the early 70s, Pascal provided the basis for various extensions such as ISO Pascal [30] and Turbo Pascal [29]. Upon introduction, Pascal facilitated *modularity* as a then novel concept, allowing for large programs to be divided into *modules*, thereby improving program comprehension [21]. We schematically illustrate the structure of a Pascal program in Fig. 1, focusing on the properties we use for our comparison.

A Pascal *program* consists of *variables*, *constants*, *data types*, *procedures*, *functions* and an *implementation*. These components are enclosed by their corresponding blocks. The program variables, for instance, are declared in an enclosing *variable declaration* block. The combination of blocks describe the resulting program to have the implementation as a final block. Representing the entry point of the program, the implementation defines the runtime behavior. It has access to all other block components acting on them through statements, such as assignments or function calls. In contrast to programs, *units* only define static components like variables or procedures, excluding an implementation. Consequently, units are not executable but can be included by programs, thus facilitating the reuse of software artifacts. Similarly, the variants

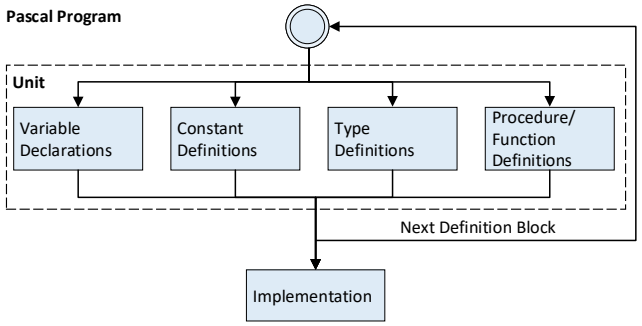


Figure 1: Standard Pascal Language Structure

in our case study structure their programs by including external units which we will refer to as *includes* in the next sections.

2.2 Family Mining

During system evolution, requirements for a new group of clients may emerge, creating a need for new variants to address them. With *clone-and-own*, a subsequent transition towards structured reuse requires variant analysis to identify their commonalities and differences [35]. *Family mining* describes a concept, in which meta-model instances, each reflecting a specific implementation, are compared using metrics to re-engineer their variability information [33, 35]. The result is a *family model* which depicts implementation artifacts with their variability information annotated [34]. To this end, variability mining is a vital step of the extractive model adoption approach for transferring a variant-rich system into an SPL [13].

We provide two Pascal program variants in Fig. 2 to illustrate *clone-and-own*. Both programs comprise two *Integer* variables *I* and *Sum* but operate differently as the *expression* within the *For loop* differs. Furthermore, line 5 and 6 have been switched between variant 1 and 2. Moreover, *Variant 2* adds an additional statement in line 12.

Variant 1:

```
1  Program SUM (Input, Output);
2  Var
3      I, Sum:Integer;
4  Begin
5      Sum := 0;
6      I := 1;
7      for i := 0 to 100 do
8        Begin
9          Sum := Sum+1;
10         I:=I+2;
11       End;
12
13 End.
```

Variant 2:

```
Program SUM (Input, Output);
Var
    I, Sum:Integer;
Begin
    I := 1;
    Sum := 0;
    for i := 0 to 50 do
      Begin
        Sum := Sum+1;
        I:=I+2;
      End;
    Writeln (Sum);
End.
```

Figure 2: Pascal Programs Evolved from *Clone-and-Own*

3 COMPARING LEGACY PASCAL PROGRAMS

In this section, we detail the meta-model we created for Pascal program-comparison, provide information on the used comparison metrics and elaborate on our comparison approach to compare Pascal-systems and identify there variability.

3.1 A Meta-Model for Pascal Programs

For imperative programming languages, such as Pascal, their textual representations may complicate the extraction of their variability information. Without taking the language structure into account, the identification of comparable strings may pose a problem. Therefore, we transform the Pascal language structure into a meta-model to allow for accurate comparison. The model-driven approach reduces the complexity of programs, facilitates the identification of variability-containing program parts, thereby promoting structured reuse of implementation artifacts. For our meta-model, we used the *Eclipse Modeling Framework (EMF)*[1] and in particular Ecore[2] meta-models. Such meta-models allow to generate data structures for our comparisons in Java[3]. In accordance to the structure of Pascal programs (cf. Fig. 1), we depict our meta-model in Fig. 3. The meta-model shown in Fig. 3 maps to the standard Pascal language structure (cf. Sec. 2). However, variable and constant definitions are collapsed within our meta-model and represented as a single *Variable* element. Furthermore, procedures and functions are collapsed and represented as a single *Operation* entity. By that, we enable future additions of further Pascal programs, which may feature different language extensions, for instance with Turbo Pascal (cf. Sec. 2). For constructing our meta-model, we identified three

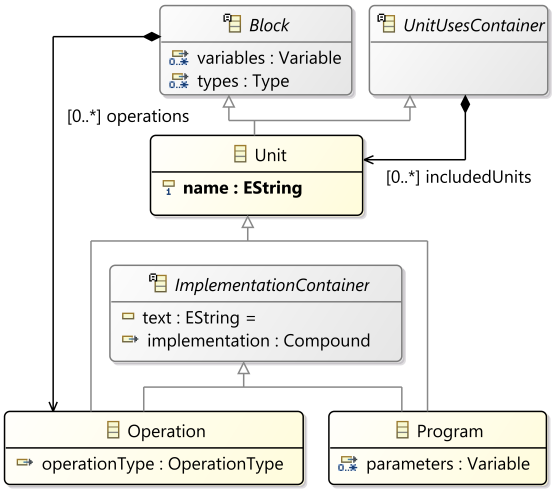


Figure 3: The Pascal Language Ecore Model

major components, which are *programs*, *units* and *operations*. Although such components share some attributes, i.e., a *name*, they all have individual details. For instance, units do not provide an implementation, while programs do contain an implementation. The *ImplementationContainer* in Fig. 3 serves as a placeholder for another meta-model to capture specific implementation details, such as individual statements, loops, and their nesting respectively. By that, we can capture different expressions, such as statements and mathematical expressions in Fig. 2. Further details on the meta-model are given online[4].

We implemented a parser based on the grammar of Pascal. However, the provided industry programs feature additional language characteristics aside from Pascal. We consequently extended the Pascal grammar accordingly. Our parser is able to compile the Intels[5] derivation [8] and further dialects within the evaluated case study, which are confidential. The parser produces an abstract syntax tree that maps to our meta-model representation.

3.2 A Customizable Metric

With our approach, it is possible to define different metrics by combining various *attributes*, which we list in Tab. 1. An *attribute* is a logical unit, which compares two elements of the meta-model instances and returns the result in the form of a similarity value between 0.0 and 1.0. Hence, our metric can be customized by composing attributes individually and adjusting their weights accordingly. Attributes are grouped in *categories* by the structure of Pascal programs. Detailed in Tab. 1, *program attributes* define attributes specifically designed for the comparison of overall properties of Pascal programs. Furthermore, *statement attributes* target specific statements and, for instance, compare assignment operators if present within the evaluated statements. Some element attributes, marked *conditional* in Tab. 1, are only evaluated for certain element types. For instances, the attribute *Constant Value Compare* applies only to *variables*, which are *constants*. This design prevents to lower the similarity of the comparison results based on a property that both elements don't have, such as variables don't have a value.

In Tab. 1, we provide 20 attributes that are grouped into four categories. By individually composing attributes from Tab. 1, different metrics can be created, which can be of varying granularity. For instance, the metric *M1* in Tab. 1 only contains *program attributes* and, thus, evaluates only overall Pascal program peculiarities, such as the number of contained statements. The metric *M2* in Tab. 1 evaluates variables, constants and types on module and procedure/-function level. It also compares all of these attributes in included Pascal units. The metric *M3* in Tab. 1 compares the implementation of operations and of the main program. We employ three metrics in total, *M1, M2,* and *M3*, each comparing a different aspect of Pascal programs and progressively increasing the level of detail. Attributes can be fine-tuned by regulation of their *weights*, which define the relevance of the attributes for the comparison result. We chose the weights based on the arithmetical average of the attributes that are used for a given property. For every category of attributes, their combined weight must be equal to one.

3.3 Comparison Approach

Our approach to compare Pascal programs comprises different phases, and we depict its overall workflow in Fig. 4. First, Pascal programs are transformed into an instance of our meta-model by

[1]Eclipse Foundation™- https://www.eclipse.org/modeling/ - May 2019
[2]Eclipse Foundation™- https://wiki.eclipse.org/Ecore - May 2019
[3]Oracle®- https://www.java.com/ - May 2019
[4]Online material: https://www.isf.cs.tu-bs.de/data/rosiak/material/variVolution19
[5]Intel Corporation®- www.intel.de - May 2019

Table 1: Pascal Program Attributes for Comparison with her Membership to a Metric and specific Weight.

Attribute Name	Metric	Weight
Program Attributes		
Include Count	M1	0.2
Procedure Count	M1	0.2
Statement Count	M1	0.2
Type Count	M1	0.2
Variable Count	M1	0.2
Variable Attributes		
Constant Value Compare (conditional)	M2	0.5
Damerau/Levenshtein Name Compare [15][2]	M2	0.5
Variable Type Compare (conditional)	M2	0.5
Type Attributes		
Damerau/Levenshtein Name Compare [15][2]	M2	0.5
Record Element Count Compare (conditional)	M2	0.5
Array Bounds (conditional)	M2	0.5
Statement Attributes		
Assignment Compare (conditional)	M3	0.5
If Condition Compare (conditional)	M3	0.5
Nested Statement Structure Compare	M3	0.5
Procedure Call Compare (conditional)	M3	0.5
With Variable Compare (conditional)	M3	0.5
Case Compare (conditional)	M3	0.5
For Statement Compare (conditional)	M3	0.5

our specialized Pascal parser. The process is divided into two parts, the *customizable* part and the *automated* part. In the *customizable* part, the user defines a metric that determines which properties of the programs are to be compared. Hence, the metric consists of a set of attributes and their individually assigned weights. The metric partitions the different attributes into the model element categories they compare. The elements are processed top-down as there is a containment relation between categories, e.g., a program contains variables. When comparing two elements, e.g., $e1$ and $e2$ in the same category, their sub-elements are compared recursively. The result of these sub-element comparisons, together with the applied attributes on $(e1, e2)$, contribute to the overall similarity of the pair. Consequently, the similarity value for two programs is also defined by the similarity of their statement if they have been evaluated during the comparison. After the definition of a metric in the customizable part, it is subject to the automated part. Here, the compare process is executed, applying the metric to identify commonalities and differences between two Pascal programs, i.e., two meta-model instances. We refer to the two instances as the *source model* and the *target model*. Both models are first disassembled into single elements and then compared with each other. This process only considers pairs of elements that are in the same category. The comparison-result is a complete and weighted bipartite graph between elements of the source model and target model. For every category, for instance, statements (cf. Fig. 3), the bipartite graph represents all possible assignments with their similarity values. Subsequently, elements from the source model and elements from the target model are *matched*. During matching, we first sort possible assignment between elements from both models in a descending order based on their similarity value. We then retrieve the assignments with the highest similarity, and for each element, assign at most one element from the other model. After matching elements, we present the result in the form of a family model [33], categorizing comparison results for elements A and B based on the thresholds given in Fig. 1 with λ being user-adjustable.

$$rel(A,B) = \left\{ \begin{array}{ll} mandatory & \lambda \leq sim(A,B) \\ alternative & 0 < sim(A,B) < \lambda \\ optional & 0 = sim(A,B) \end{array} \right\} \quad (1)$$

4 CASE STUDY

In this section, we state our research questions and provide information on the case study used to demonstrate the feasibility of our approach. We also detail the metrics we used for the assessment.

4.1 Research Questions

With our approach, we compare two Pascal module variants to identify common and varying parts within them. We focus on the following research questions:

RQ1: *Can we identify common and varying parts within programs?* We assess to what extent our metrics can capture common and varying parts between two Pascal programs.

RQ2: *Can we consider performance of our approach reasonable?* An acceptable runtime is pivotal, especially in industry. We refer to performance as the total runtime required to compare two programs and to match their elements.

Furthermore, we report on our experiences and peculiarities we identified for the case study, which directly relates to their evolution-history and discuss our findings mention.

4.2 Setup

We evaluated our approach on an Intel Core i7-3770k (3,5 GHz) with 16 GB of RAM, running Windows[6] 10 on 64bit. We implemented our approach in Java as part of a toolkit, which, in principle, can be extended for the analysis of further languages.

Our case study comprises seven variants of a control unit module, totaling over 13k LOC. The modules provided by industry partner are confidential. The initial creation of the modules was in 1996. However, the date they are modified last varies due to the evolution. We list respective information in Tab. 2. Each module variant was developed for a specific project with different requirements.

4.3 Metrics Used for Comparison

We define three metrics with different granularity to compare Pascal programs and to identify common and varying parts within them. Tab. 1 states contained attributes for each metric *M1*, *M2* and *M3*. A full description of the attributes is provided on our website[4].

[6] Windows® - https://www.microsoft.com/windows - May 2019

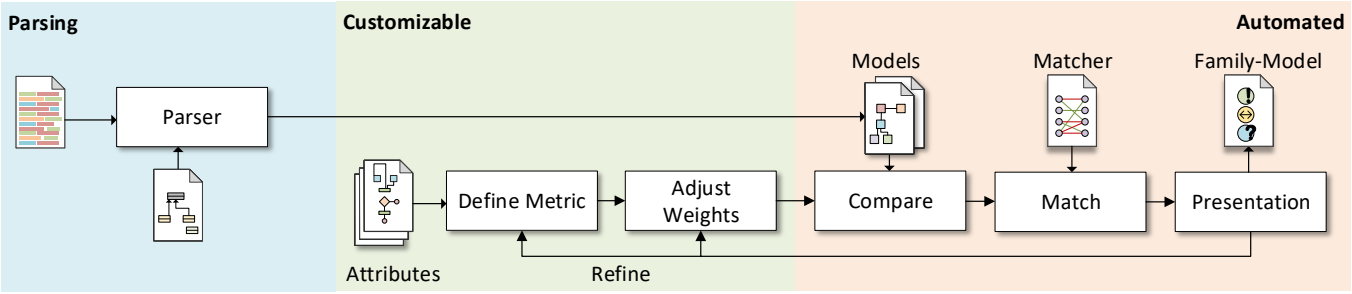


Figure 4: Workflow of our Approach to Compare Pascal Programs

Table 2: Pascal Module Variants

Name	#Lines	#LOC	#Comm.	LOCC	#Ann.LOC	mod. date
V1	4757	1789	2774	2408	53	2017
V2	4987	1875	2909	2529	53	2014
V3	2870	1914	766	0	44	2000
V4	2850	1904	745	0	44	1998
V5	2859	1911	749	0	44	2004
V6	2870	1914	766	0	44	2000
V7	2936	1964	765	0	44	2011

#Lines – Total Number of Lines, LOC – Lines of Code, #Comm. – Comments, LOCC – Lines of control characters, Ann. – Annoted, mode.

M1 (Count Metric): The metric compares the structure of a program using attributes such as the count of variables or operation count. The similarity value for an attribute represents the extent to which compared elements share the evaluated property. Consequently, it will be calculated as follows.

$$similarity = \frac{min(sc, tc)}{max(sc, tc)} : sc, tc \in \mathbb{R}_{>0}$$

with *sc* being the source model count and *tc* being the target model count.

The metric *M1* does not allow for the identification of changes within the internal structure of elements, e.g., *renamed variables*. Hence, we create the *structure metric (M2)*.

M2 (Structure Metric): With the attributes defined for *M2*, we compare the definition of a module, functions, procedures, and all includes. Therefore, we can match the respective elements with higher precision than with *M1*, as we consider internals of individual elements and not only higher level properties such as variable count. Name and type changes of variable and type definition can be detected. The attributes *Record Element Count* and *Array Bounds Compare* in particular, are specialized for a specific data type. The former is applied to record definitions that alias the composition of records, while the latter only compares array definitions.

M3 (Static Implementation Metric): With *M3*, we can analyze statements and their composition. When evaluating individual statements, *M3* allows to compare the nested statements, e.g., if and else blocks. This process continues to compare sub-statements of the source statement with sub-statements of the target statement as long as there are nested statements to compare. For this reason, the attributes are responsible only for the internal properties, while recursion propagates similarities upwards. The metric *M3* allows for the detection of clones and changes on an implementation level. Every attribute in *M3*, except *Nested Statement Structure*, implements a special comparison algorithm designed for its unique kind of statement.

5 RESULTS AND DISCUSSION

In this section, we present and discuss the results produced by our approach. We show aggregated information in this section and refer to our supplementary material[4] for further details.

RQ1: Identification of Common and Varying Parts: We applied all pairwise comparisons of our case study the three metrics (cf. Sec. 4) we defined. For the metric *M1*, we provide the similarity values calculated for individual module variant comparisons in Tab. 3. Overall, the similarities range from 96.7% to 99.9%. For *M1*, we found every module variant to exhibit the same number of *includes*, i.e., referenced units, and also have the same number of *procedures/functions*. Further evaluation revealed that changes are in included units, such as additional declarations of *variables*. For instance, *V4*, *V5* and *V6* have more variables defined in the main program than other variants. In general, the results in Tab. 3 hint towards a high similarity between all module variants in general. Moreover, we used our more fine-grained metric *M2* to get detailed

Table 3: Similarities of Pascal Modules using *M1*

%	V1	V2	V3	V4	V5	V6	V7
V1	╲	99.0	99.4	98.4	98.0	99.3	97.6
V2		╲	99.0	97.6	97.1	99.0	96.7
V3			╲	98.6	98.1	**99.9**	97.6
V4				╲	99.6	98.4	99.4
V5					╲	98.0	99.4
V6						╲	97.5
V7							╲

information about the changes in the included units and the modules themselves. In Tab. 4, we show similarity values for all module comparisons with the *M2* metric. Here, the similarity value ranges from 85.77% to 99.46%. Overall, the similarity values have decreased compared to the results we achieved with the metric *M1* (cf. Tab. 3). With the metric *M2*, we were able to substantiate our previous findings and *M2* revealed differences between included units. More precisely, we detected that data types of variables changed between module variants, and further variables were added.

Furthermore, we utilized the metric *M3* for all pairwise comparisons, yielding the similarity values listed in Tab. 5. Analyzing

Table 4: Similarities of Pascal Modules using M2

%	V1	V2	V3	V4	V5	V6	V7
V1	╲	97.1	97.0	89.7	89.3	96.4	88.8
V2		╲	95.3	88.0	87.7	94.8	87.2
V3			╲	92.6	92.3	98.4	90.7
V4				╲	**99.6**	91.1	98.0
V5					╲	90.7	98.4
V6						╲	89.9
V7							╲

individual statements revealed changes in the implementation of procedures and the main implementation. Most of these changes are added conditions or changed returning variables. More precisely, we found all procedures/functions to be equal except *V4*, *V5* and *V7*, which extended functions/procedures by further variables. Due to the confidentiality of the evaluated Pascal module

Table 5: Similarities of Pascal Modules using M3

%	V1	V2	V3	V4	V5	V6	V7
V1	╲	99.9	97.0	87.7	88.2	97.0	88.2
V2		╲	97.0	87.7	88.2	97.0	88.2
V3			╲	90.7	91.2	**100.0**	91.2
V4				╲	98.6	90.7	98.6
V5					╲	91.2	99.9
V6						╲	91.2
V7							╲

variants, we can not show a family model with actual data. However, we applied our approach to the Pascal variants from Fig. 2 and depict the created family model in Fig. 5. By that, we illustrate similar and variability-containing parts to be captured, as well as optional parts, such as the *Writeln* statement,to be only present in Variant 2 from Fig. 2. We argue that our approach and

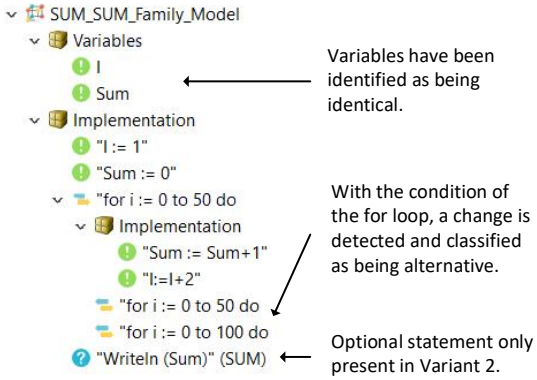


Figure 5: Family Model for the Pascal Variants from Fig. 2

the metrics we defined allow to capture even small changes such as altered *conditions* within individual *statements*. We note that by using the metric *M3*, the variants *V3* and *V6* has an equal implementation. One other finding is that the variants *V4*, *V5*, *V6* and *V1*, *V2*, *V3*, *V6*, have identical procedure/function implementations. Most of the changes that we can determine within the implementation are added cases or changed conditions. Utmost of the results were presented to our industrial partners to assess it with expert knowledge because no ground truth exists for this real system.

RQ2: Scalability of our Approach: We measure performance using an event-based benchmark system implemented in our toolkit. This system allows it to start and stop runtime measurement in any phase of the comparison process. The memory consumption value is based on the windows resource monitor. To reduce deviations such as compiler optimization, we measured every runtime ten times and calculated the average value. Tab. 6 provides all runtime results for all pairwise comparisons using the metric *M2*, given in minutes. The results show that our approach compares the smaller modules *V6* and *V7* with ≈ 4,000 LOC and 1281 included variables/constants and types in under one minute. The comparison of V1 and V2 reaches the maximum measured runtime. The reason is that the number of variables, constants, and types in the included files is higher for these variants. Such elements are processed during our approach, thereby increasing overall runtime. To further evaluate scalability, we assessed the number

Table 6: Total Runtime for Pairwise Comparisons

min.	V1	V2	V3	V4	V5	V6	V7
V1	╲	2:23	2:04	2:05	2:02	1:02	2:03
V2		╲	2:03	2:02	2:00	1:08	2:01
V3			╲	1:59	1:57	1:00	1:59
V4				╲	1:58	1:00	1:56
V5					╲	1:00	1:54
V6						╲	0:57
V7							╲

of comparisons performed for individual elements. By that, we aim to relate the number of comparisons to the overall runtime. Detailed information for the comparison of *V1* and *V2* is given in Tab. 7. For instance, Tab. 7 shows five attributes used for comparing *V1* and *V2*, therefore resulting in one possible pair. Hence, with each attribute evaluated separately for such a pair, a total of five comparisons is performed. Furthermore, over 62,000 pairs of statements are created. Evaluating each pair of statements with seven attributes results in over 435,000 comparisons. Overall for *V1* and *V2*, around 737,000 element pairs are created, resulting in around 2,5 million comparisons. However, relating such number of comparisons to the respective runtime in Tab. 6 with ≈ 2,5 minutes and only usage of 784 MB of memory, we argue that our approach using a detailed metric, which analyses individual statements does scale even for large systems. Moreover, we evaluated the distribution of runtime for the *Comparison* and *Matching* phases of our approach (cf. Fig. 4). For the comparison of *V1* and *V2*, we provide respective results in Tab. 8. With ≈ 77%, the *Comparison* makes up

Table 7: Pairs of Elements and Comparisons for V1 & V2

Elements	Pairs	Attributes	Comparisons
Programs	1	5	5
Variables	667.522	3	2.002.566
Types	7.864	3	23.592
Statements	62.267	7	435.869
Overall	737.654		2.462.032

Table 8: Time Measurement of Comparison of V1 with V2

Process	AVG Runtime in seconds
Compare Time	102,915
Matching Time	30,891
Total Time	133,806

most of the required runtime. During the comparison, we compare all possible pairwise combinations. However, during matching, we sort results based on their similarity value and retrieve pairs in descending order with higher values first. Consequently, the *Matching phase*, requiring less time than the *Comparison phase*, is reasonable.

Discussion of the Case Study. The given control unit variants have a common core, which had then been adapted to comply with the demands of different customers, thereby specific project requirements. Moreover, we found each module to consist of a main file acting as an entry point and a set of *include statements*, which include *constant* and *type* definitions. Apart from Pascal as defined in [8], the evaluated modules contains extended language concepts. Specifically, we identified special annotations in the code, which serve to control the overall program execution. Furthermore, we identified special *control characters*, which we found to originate from an automated code-generation tool, applied in the past for generating Pascal code from graphical charts. The modules are compiled by an adjusted Pascal86 compiler that is able to handle annotations. Moreover, we found the file naming to be consistent across all variants and the main program to include the same units.

However, within the module files themselves, there are notable differences. We detected renamings due to changes in natural language, specifically translations from German to English. We noted this, as the name of a variable changed but the value and the usage of the variable itself remained unchanged. In some instances, renaming was explicitly stated within comments in the modules. Moreover, we identified naming conventions, which we found to be problematic for comparison. With a certain pattern being static, metrics that calculate the distance between two strings, such as Damerau-Levenshtein distance[15][2], may produce results practitioners may find incomprehensible. The resulting base similarity cause elements to be classified as alternatives in the family model. A way to address this is to weight attributes with distance metrics less than other attributes.

Upon evaluation of variables and procedures in the main files of the module variants, we found little to no change within two groups of modules: *V1*, *V2*, *V3*, *V6* and *V4*, *V5*, *V7*. In both groups, variables and functions are identical. The only differences in these groups are due to the included definition files.

However, for modules from different groups, the results indicate significant differences. We found not only changes to the functionality of the code but also to the programming conventions themselves. It is worth noting that due to the extensive development period and the fact that multiple teams participated in that process, a new variant can introduce new programming conventions and style. For example, the language for variable and type names varies in files and variants between English and German. Additionally, the meta-data, such as comments, was found to be an important aspect. Whether it is descriptions, control characters or versioning information, we found meta-data to occupy a great portion of the file as shown in Tab. 2. Specifically, we found control characters in *V1* and *V2*, which are absent in all the other variants. For remaining module variants, we noticed a corresponding drop in lines of comments (cf. Tab. 2). Both are generic variants that serve as a basis for new ones trough clone-and-own. The other variants have either *V1* or *V2* as their base or they branched off from another variant due to varying requirements.

Nevertheless, the similarity values reveal a very high similarity between the module variants in general. Two variants have been identified to be exact clones of each other. However, developers were not aware of this and did maintain both module variants independently from each other.

6 RELATED WORK

With our approach, we compare Pascal modules to identify common and varying parts. Our approach relates to two main categories, which are *Clone Detection* and *Variability Analysis*.

Clone Detection and Diffing. Different code clone detection approaches shown in the past years. However, most of them are based on the source code, while only a few based on models. An approach to compare pairs of *Unified Modeling Language (UML)* class diagrams is proposed in [11], semantically lifting derived differences to enhance comprehensibility. They extend their work in [10] and focus on *State Charts*. However, unlike our approach, results are not transferred into a family model, in which we classify results. In [36] is showed how to detecting code clones in Java using the Smith-Waterman algorithm on bytecode. This approach is only suitable for languages that translated into Java-Bytecode. In contrast to Java, Pascal compiles into machine language and is executed directly and not interpreted by a Java Virtual Machine, so this approach is not suitable. In [7], an incremental graph-based clone detection algorithm is presented. By employing a data structure called the *clone index*, subgraphs are stored and labeled for efficient matching. Although the containment relation between statements forms a graph, adding or removing a statement alters the underlying graph. Consequently, the detection is unsuitable for variability mining, as we want to quantify potential clones in terms of concrete similarities. [9] shows a token-based approach. In contrast, our technique allows identifying inserted and modified code-fragments.

Variability Analysis. The identification of variability in software evolution has been subject to research for several years. However, there are few extractive approaches focused on reverse engineering without feature information. In [27], an adjustable matching window technique is employed to enhance variability analysis in *Matlab/Simulink* models within the Family Mining Framework[33]. The windows define a subgraph on which data flow based comparison is applied for each model. The approach takes hierarchical compositions of blocks into account while comparing windows, contrary to ours in which statements strictly compared within the same level. *Variant Analysis* [6] is an approach for analyzing multiple software variants individually that mapped onto a system structure model. Although the variants represented as data models, they use a string-based algorithm for the comparison of source code. As our data model stores elements with their corresponding string within the source code, we support string-based comparisons as well. However, our approach operates on pairs of software variants and does not merge the analysis results since we compare on a low level of detail.

In [28] reverse-engineers feature models from product maps. Our approach generates a family model based on code comparison results. In [17, 37] it is shown how to identify variability on existing model variants. Both approaches only work for models and are not suitable for source-code.

7 CONCLUSION AND FUTURE WORK

Clone-and-own prevails as a reuse strategy that results in a proliferation of redundant and similar but non-alike assets. With proper documentation not cherished, maintainability of the emerging portfolio is adversely affected, and resources may be deferred from ongoing system evolution. A system analysis is required to identify common and variability-containing parts to reinstate sustainable development. For legacy software systems, this remains an open challenge. In this paper, we have addressed this issue and introduced a model-based comparison approach and several metrics for Pascal modules. We have demonstrated the feasibility of our technique using an industrial case study comprising seven Pascal modules totaling over 13,000 lines of code. We have shown our approach to apply to such Pascal systems and to scale even for the comparison of large modules. Furthermore, we highlighted and discussed peculiarities we identified for the case study, given its long evolution history. We identified natural languages to vary between implementations, different conventions due to different teams involved, and various code generation tools. With our approach, we create a family model, which captures similar and variability-containing assets for Pascal modules. In future work, we will extend upon our evaluation by assessing further case studies. By that, we aim to identify further metrics and current limitations of our approach. Furthermore, we plan to discuss our findings with developers to assess the precision of our approach, as well as to identify further use cases for our approach. In addition, we want to extend our work with an iterative approach to compare n-models. By that, we aim to getting a variability model that represents the variability of multiply software variants.

ACKNOWLEDGMENT

This work has been partially supported by the German Research Foundation (DFG) (SCHA 1635/12-1).

REFERENCES

[1] W. J. Cullyer and B. A. Wickmann. 1991. The choice of computer languages for use in safety-critical systems. *Software Engineering Journal* 6, 2 (March 1991), 51–58.
[2] F.J Damerau. 1964. A technique for computer detection and correction of spelling errors. *Commun. ACM* 7, 3 (1964), 171–176.
[3] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz. 2010. Model Clone Detection in Practice. In *Proc. of the Intl. Workshop on Software Clones (IWSC)*. ACM, 57–64.
[4] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *Proc. of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34.
[5] Z. Durdik, B. Klatt, H. Koziolek, K. Krogmann, J. Stammel, and R. Weiss. 2012. Sustainability guidelines for long-living software systems. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 517–526.
[6] S. Duszynski, J. Knodel, and M. Becker. 2011. Analyzing the Source Code of Multiple Software Variants for Reuse Potential. In *2011 18th Working Conference on Reverse Engineering*. 303–307.
[7] B. Hummel, E. Juergens, and D. Steidl. 2011. Index-based Model Clone Detection. In *Proceedings of the 5th International Workshop on Software Clones (IWSC '11)*. 21–27.
[8] Intel Corporation 1985. *Pascal-86 User's Guide*. Intel Corporation. Order Number 121539-003.
[9] T. Kamiya, S. Kusumoto, and K. Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.
[10] M. Kelter, U.and Schmidt. 2008. Comparing State Machines. ACM, 1–6.
[11] U. Kelter, J. Wehren, and J. Niere. 2005. A Generic Difference Algorithm for UML Models. *Software Engineering* 64, 105-116 (2005), 4–9.
[12] J. A. Kim. 2010. Case Study of Software Product Line Engineering in Insurance Product. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. Springer, 495–495.
[13] C. Krueger. 2002. Easing the Transition to Software Mass Customization. In *Software Product-Family Engineering*, F van der Linden (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 282–293.
[14] R. Lapeña, M. Ballarin, and C. Cetina. 2016. Towards Clone-and-own Support: Locating Relevant Methods in Legacy Products. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 194–203.
[15] V.I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.
[16] F. J. van der Linden, K. Schmid, and E. Rommes. 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer.
[17] J. Martinez, T. Ziadi, T. F Bissyande, J. Klein, and Y. Le Traon. 2015. Automating the extraction of model-based software product lines from model variants (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 396–406.
[18] M.Schulze, J.Mauersberger, and D. Beuche. 2013. Functional Safety and Variability: Can it be brought together?. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM.
[19] Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Complete and Accurate Clone Detection in Graph-based Models. In *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE.
[20] K. Pohl, G. Böckle, and Linden, F. J. van der. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.
[21] G.P. Radha. 1999. *Pascal Programming*. New Age International (P) Limited.
[22] C. Riva and C. Del Rosso. 2003. Experiences with Software Product Family Evolution. In *Proc. of the Joint Workshop on Software Evolution and Intl. Workshop on Principles of Software Evolution (IWPSE-EVOL)*. IEEE, 161–169.
[23] J.S. Rohl and H.J. Barrett. 1990. *Programming Via Pascal*. Cambridge University Press.
[24] J. Rubin and M. Chechik. 2012. Combining Related Products into Product Lines. In *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 285–300.
[25] J. Rubin and M. Chechik. 2013. Quality of Merge-Refactorings for Product Lines. In *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 83–98.
[26] A. Schlie, D. Wille, L. Cleophas, and I. Schaefer. 2017. Clustering Variation Points in MATLAB/Simulink Models Using Reverse Signal Propagation Analysis. In *Proc. of the Intl. Conference on Software Reuse (ICSR)*. Springer, 77–94.
[27] A. Schlie, D. Wille, S. Schulze, L. Cleophas, and I. Schaefer. 2017. Detecting Variability in MATLAB/Simulink Models: An Industry-Inspired Technique and Its Evaluation. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 215–224.
[28] S. She, R. Lotufo, A. Berger, T.and Wąsowski, and K. Czarnecki. 2011. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 461–470.
[29] G.B. Shelly, T.J. Cashman, and S.G. Forsythe. 1987. *Turbo Pascal programming*. Boyd & Fraser.
[30] SO/IEC 1991 1991. *Pascal ISO 7185:1990*. SO/IEC 1991.
[31] K. J. Sullivan, W.G. Griswold, Y. Cai, and B. Hallen. 2001. The Structure and Value of Modularity in Software Design. In *Proc. of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 99–108.
[32] M. T. Valente, V. Borges, and L. Passos. 2012. A Semi-Automatic Approach for Extracting Software Product Lines. *IEEE Transactions on Software Engineering* 38, 4 (July 2012), 737–754.
[33] D. Wille. 2014. Managing Lots of Models: The FaMine Approach. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 817–819.
[34] D. Wille. 2019. *Custom-Tailored Product Line Extraction*. Ph.D. Dissertation.
[35] D Wille, D Tiede, S Schulze, C Seidl, and I Schaefer. 2016. Identifying Variability in Object-Oriented Code Using Model-Based Code Mining. In *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, Tiziana Margaria and Bernhard Steffen (Eds.).
[36] D. Yu, J. Yang, X. Chen, and J. Chen. 2019. Detecting Java Code Clones Based on Bytecode Sequence Alignment. *IEEE Access* 7 (2019), 22421–22433.
[37] X. Zhang, Ø. Haugen, and B. Moller-Pedersen. 2011. Model comparison to synthesize a model-driven software product line. In *2011 15th International Software Product Line Conference*. IEEE, 90–99.

A Process for Fault-Driven Repair of Constraints Among Features

Paolo Arcaini
National Institute of Informatics
Japan
arcaini@nii.ac.jp

Angelo Gargantini
University of Bergamo
Italy
angelo.gargantini@unibg.it

Marco Radavelli
University of Bergamo
Italy
marco.radavelli@unibg.it

ABSTRACT

The variability of a Software Product Line is usually both described in the *problem space* (by using a *variability model*) and in the *solution space* (i.e., the *system implementation*). If the two spaces are not aligned, wrong decisions can be done regarding the system configuration. In this work, we consider the case in which the variability model is not aligned with the solution space, and we propose an approach to automatically repair (possibly) faulty constraints in variability models. The approach takes as input a variability model and a set of combinations of features that trigger conformance faults between the model and the real system, and produces the repaired set of constraints as output. The approach consists of three major phases. First, it generates a test suite and identifies the condition triggering the faults. Then, it modifies the constraints of the variability model according to the type of faults. Lastly, it uses a logic minimization method to simplify the modified constraints. We evaluate the process on variability models of 7 applications of various sizes. An empirical analysis on these models shows that our approach can effectively repair constraints among features in an automated way.

CCS CONCEPTS

• **Software and its engineering → Software product lines**;

KEYWORDS

automatic repair, fault, variability model, system evolution

ACM Reference Format:
Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. 2019. A Process for Fault-Driven Repair of Constraints Among Features. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3307630.3342413

1 INTRODUCTION

Most software systems can be configured in order to improve their capability to address users' needs. Configuration of such systems is generally performed by setting system parameters [34]. These parameters, or *features*, can be identified at design time. For instance, in the case of a *software product line*, the designer identifies the features unique to individual products and features common to all products. Such options can also be decided during compilation time, in order to improve some characteristics of the compiled code (scalability, efficiency, etc.) or to activate/deactivate some functionalities. For example, in the case of preprocessor directives, the programmer can decide which libraries to use, what code to execute and what to ignore etc. Software configurations can also be modified during operation time, when the system is already running. In this case, for example, the parameters can be saved in a configuration file and modified if necessary. Such a configuration file can also be used to decide which features to load at startup.

Model $\mathcal{M}$	Implementation (System $\mathcal{S}$):

$A \rightarrow B$
$A \rightarrow C$
	`#ifdef C //Hello`
	`char* msg = "Hello…!";`
	`#endif`
	`#ifdef B // Bye`
	`char* bye = "Bye";`
	`#endif`
	`#ifdef A // lowercase`
	`msg [0] = 'h';`
	`bye [0] = 'b';`
	`#endif`

Figure 1: Example of problem and solution spaces

Constraints exist among system features. They can prohibit system configurations that are dangerous or undesired, or can describe conditions leading to certain properties or errors in code, such as *preprocessor errors*, *parser errors*, *type errors*, and *feature effect* [30]. Designers, developers, and testers can greatly benefit from modelling features and constraints among them, as it allows to reduce development effort [33] and to identify corner cases of the system under test.

Constraints among features can be modeled using *variability models*, and imposed on the implementation by means of preprocessor directives, makefiles, etc. These two ways of modeling variability are usually known as *problem space* and *solution space* [30]. Fig. 1 presents an example of a variability model containing the constraints among three system features (A, B, and C) that are implemented as preprocessor directives in the C program.

This separation between problem and solution space allows users to model configuration without knowledge about low-level implementation details. On the other hand, these two spaces need to be consistent; code and models, however, are often not kept synchronized, and *repairs* are needed. In the evolution of product lines, two common types of repair are performed: *debugging and program repair*, when the variability model is correct but the implementation has to be fixed; and *model repair*, when the program is correct,

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342413

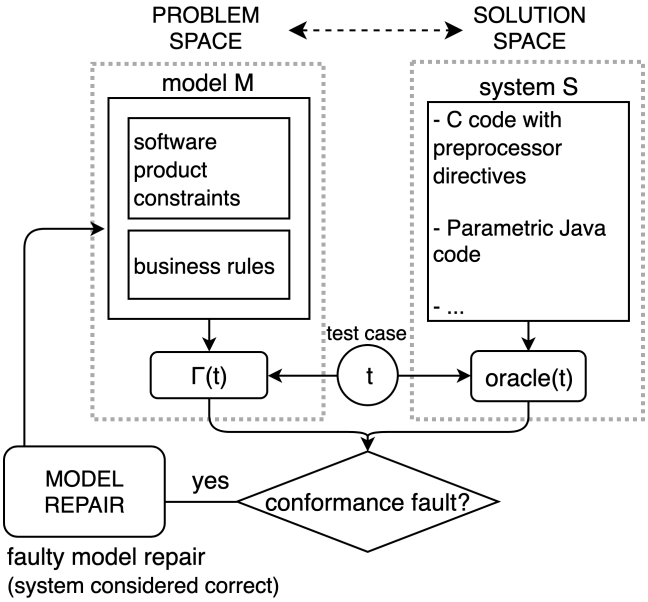


Figure 2: Fault-driven repair of variability models

but the variability model is outdated. This latter case occurs when the description of variability is evolved in the implementation, but not in the variability model. This work tackles this problem and proposes a technique to automatically repair variability models. Fig. 2 shows an overview of this context.

In order to detect discrepancies between the problem space and the solution space, classical techniques for testing of propositional formulas (as the constraints of a variability model) can be used: the classical decision and condition coverage, the MCDC [17], fault based criteria [9], and also combinatorial testing [10]. In this paper, we assume that some tests have been generated according to some coverage criterion and some *faults* (i.e., non-conformances of the model w.r.t. the solution space acting as oracle) have been detected; our aim is to *repair* the constraints of the variability model in order to remove the faults. We propose an automated process that, upon some failing tests, is able to *automatically* correct the constraints in such a way that they maintain their original validity for all the configurations, except for those found failing.

The rest of the paper is organized as follows. Sect. 2 presents some basic definitions. Sect. 3 presents the basic repair process and some possible optimizations. Sect. 4 shows the empirical results. Threats to validity are tackled in Sect. 5, whereas an overview of the related work is given in Sect. 6. Sect. 7 concludes the paper and proposes lines for future research.

2 BASIC DEFINITIONS

Definition 1 (Variability Model). *A variability model $\mathcal{M}$ is made of a set of features $F = \{f_1, \ldots, f_n\}$ and a set of constraints $\Gamma = \{\gamma_1, \ldots, \gamma_m\}$ over the features.*

The features F represent the system parameters. The expressions in Γ identify the features configurations for which the actual system is expected to work.

Definition 2 (Configuration). *A* configuration *(or* test*) t is a particular assignment of values for all the features F. We identify with $t(f_i)$ the value of feature f_i in test t. A configuration is valid if it respects the constraints, i.e., $t \models \Gamma$. We also use Γ as predicate to check the constraint satisfaction: $\Gamma(t) = true$ iff $t \models \Gamma$.*

Definition 3 (Test suite). *A* test suite *T is a set of tests. We identify with T_e the* exhaustive test suite*, i.e., the set of all the possible tests.*

Definition 4 (Oracle). *The oracle function $oracle(t)$ tells whether the configuration t is functionally correct for the system S.*

We assume that an oracle exists, that tells whether a configuration is valid or not in the real system.

We assume that the set of features F is known and correctly modeled, while the constraints could be faulty.

Definition 5 (Model correctness). *We say that the model $\mathcal{M}$ is* correct *if it conforms with the oracle for every possible configuration t, i.e., $\forall t \in T_e \colon \Gamma(t) = oracle(t)$.*

Definition 6 (Conformance fault). *We say that the model contains a* conformance fault *if there exists a configuration t such that $\Gamma(t) \neq oracle(t)$.*

Definition 7 (Combination). *A* combination *(or* partial configuration*) c is an assignment to a subset $features(c)$ of all the possible features F, i.e., $features(c) \subseteq F$. A configuration (or test) is thus a particular combination in which $features(c) = F$. The value assigned by the combination c to the feature f is denoted as $c(f)$.*

Definition 8 (Propositional representation of combinations). *A combination c can be expressed in propositional logic by making the conjunction of the truth value assignments of its features:*

$$c = \left(\bigwedge_{\{f \in features(c) | c(f)\}} f \right) \wedge \left(\bigwedge_{\{f \in features(c) | \neg c(f)\}} \neg f \right)$$

Definition 9 (Combination containment). *A test (or configuration) t* contains *a combination c if all features values in c are the same in t. Formally, $\forall f_i \in features(c) \colon c(f_i) = t(f_i)$.*

Given a test suite T, we identify all the tests containing a combination c as $T(c)$. Formally, $T(c) = \{t \in T | c \subseteq t\}$.

Definition 10 (Combination completeness). *Given a test suite T and a combination c, we say that c is* complete *w.r.t. T iff $T(c)$ contains all possible tests containing c.*

Lemma 2.1. *If c is complete w.r.t. a test suite T, then it holds $\forall t \in T_e \setminus T(c) \colon c = false$.*

Definition 11 (Failure-containing combination). *A combination c is a* failure-containing combination *(fcc) if:*

(1) c is contained in at least a failing test of T, i.e., $\exists t \in T(c) \colon \Gamma(t) \neq oracle(t)$.
(2) every configuration containing c has the same value in the oracle, i.e., $(\forall t \in T(c) \colon oracle(t) = false) \vee (\forall t \in T(c) \colon oracle(t) = true)$. In the former case, we call c an under-constraining *fcc; in the latter case, an* over-constraining *fcc.*

We further classify a conformance fault as under-constraining fault *if it exposed by an under-constraining fcc, or as* over-constraining fault *if it is exposed by an over-constraining fcc.*

Table 1: Test suites with faults (in gray)

(a) under-constraining fault

A	B	C	$\mathcal{M}_{f1}$	oracle
T	T	F	T	F
T	F	F	F	F

(b) over-constraining fault

A	B	C	$\mathcal{M}_{f2}$	oracle
F	T	T	T	T
F	F	T	T	T
F	T	F	F	T
F	F	F	F	T

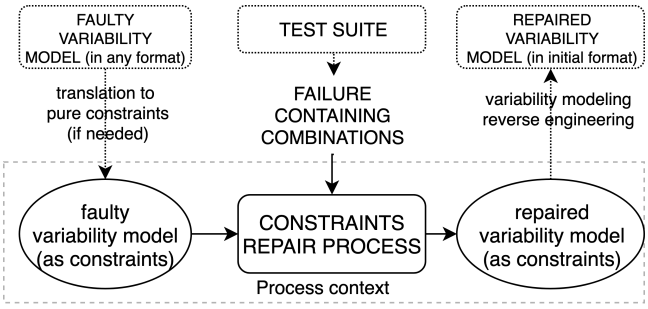


Figure 3: Context of the process to repair constraints among features in variability models

In the following, we only consider complete *fccs*.

Example 1 (Under-constraining *fcc*). Let's assume that the oracle is the system (C program) of Fig. 1 with features $F = \{A, B, C\}$ and that we have generated the test suite shown in Table 1a. Given a faulty model $\mathcal{M}_{f1}$, with only one constraint $\Gamma = \{A \rightarrow B\}$, we observe only one fault which is represented by the *fcc* $c = A \wedge \neg C$. Note that c is a complete *fcc* that identifies an under-constraining fault, i.e., it proves that the model is under-constrained.

Example 2 (Over-constraining *fcc*). Consider now a faulty version $\mathcal{M}_{f2}$ of the model in Fig. 1, characterized by $\Gamma = \{A \rightarrow B, C\}$. Given the test suite shown Table 1b, we detect two over-constraining faults identified by the complete *fcc* $c = \neg A$.

3 FAULT-DRIVEN REPAIR

We here propose a process to *repair* the constraints of a variability model, based on the detection of conformance faults between the model and the system, represented as failure-containing combinations. Fig. 3 shows the context in which our process is applied. We assume that a possibly faulty variability model is translated to a set of boolean formulas representing the constraints. For example, if the model is a feature model, semantic transformations presented in [12] can be used. From a sufficiently large test suite, complete *fccs* have been identified. To this aim, one can use well-known fault localization techniques like [8, 20]. Our process takes as input the *fccs* and the constraints and repair them. If the user wants to go back to the initial format of the variability model, (s)he must apply some reverse engineering (which is out of the scope of this work).

We first describe a naïve implementation of the repair process in Sect. 3.1, and we then introduce some optimizations in Sect. 3.2.

3.1 Naïve repair approach

In Def. 11, we distinguish between two types of failure-containing combinations (i.e., under-constraining and over-constraining *fcc*), depending on how the model fails with respect to the oracle.

We can devise a naïve repair approach that applies a specific type of repair on the base of the fault type:

(1) `Strengthening repair`: in case of under-constraining *fcc* c, $\neg c$ is added as a new constraint to Γ, i.e., the constraints set Γ' of the repaired model becomes $\Gamma' := \Gamma \cup \{\neg c\}$.
(2) `Weakening repair`: in case of over-constraining *fcc* c, c is disjuncted with every constraint in Γ, i.e., the constraints set Γ' of the repaired model becomes $\Gamma' = \cup_{\gamma_i \in \Gamma}\{\gamma_i \vee c\}$.

Example 3 (Strengthening repair). The *under-constraining* fault in Ex. 1 is repaired by adding $\neg c = \neg(A \wedge \neg C) \equiv A \rightarrow C$ as a new constraint in Γ. The repaired constraints become $\Gamma' = \{A \rightarrow B, A \rightarrow C\}$.

Example 4 (Weakening repair). The *over-constraining fault* in Ex. 1 is repaired by adding $c = \neg A$ in disjunction with all the existing constraints, so that the repaired constraints become $\Gamma' = \{(A \rightarrow B) \vee \neg A, C \vee \neg A\}$. Note that the first constraint is *redundant*, as it is equivalent to the original constraint $A \rightarrow B$. The only necessary application of the repair is the one in the second constraint, as it correctly allows to have both features A and C assigned to *false* (as in the oracle).

Theorem 1 (Correctness of the naïve approach). *If a combination c is complete w.r.t. its test suite $T(c)$, the repairs applied by the naïve approach to Γ (obtaining the modified constraints set Γ') are correct, i.e., they remove all existing faults in $T(c)$ and do not introduce new ones, i.e.,*

(1) $\forall t \in T(c)\colon \Gamma'(t) = oracle(t)$;
(2) $\forall t \in T_e \setminus T(c)\colon \Gamma'(t) = \Gamma(t)$.

Proof. Let's consider the two kinds of repairs separately:

- `Strengthening repair`: the repaired constraints are $\Gamma' = \{\gamma_1, \ldots, \gamma_m, \gamma_{m+1}\}$, where $\gamma_{m+1} = \neg c$.
  (1) From the definition of under-constraining *fcc*, we know that it holds $\forall t \in T(c)\colon oracle(t) = false$. Furthermore, we also know that $\forall t \in T(c)\colon \Gamma'(t) = false$, because the new constraint $\gamma_{m+1} = \neg c$ falsifies all the tests containing the *fcc* c. Therefore, it holds $\forall t \in T(c)\colon \Gamma'(t) = oracle(t)$.
  (2) By Lemma 2.1, we know that $\forall t \in T_e \setminus T(c)\colon c = false$. Therefore, the added constraint $\gamma_{m+1} = \neg c$ is always true in tests $T_e \setminus T(c)$. Since γ_{m+1} has no influence on the evaluation of these tests, it holds $\forall t \in T_e \setminus T(c)\colon \Gamma'(t) = \Gamma(t)$.
- `Weakening repair`: the repaired constraints are $\Gamma' = \{\gamma_1 \vee c, \ldots, \gamma_m \vee c\}$.
  (1) From the definition of over-constraining *fcc*, we know that it holds $\forall t \in T(c)\colon oracle(t) = true$. Furthermore, we also know that $\forall t \in T(c)\colon \Gamma'(t) = true$, because all the constraints $\gamma_i' = \gamma_i \vee c$ admit all the tests containing the *fcc* c. Therefore, $\forall t \in T(c)\colon \Gamma'(t) = oracle(t)$.
  (2) By Lemma 2.1, we know that $\forall t \in T_e \setminus T(c)\colon c = false$. Since c is added as a disjunction to the existing constraints, it leaves the constraints equivalent to the original ones, i.e., $\forall t \in T_e \setminus T(c)\colon \Gamma'(t) = \Gamma(t)$.

□

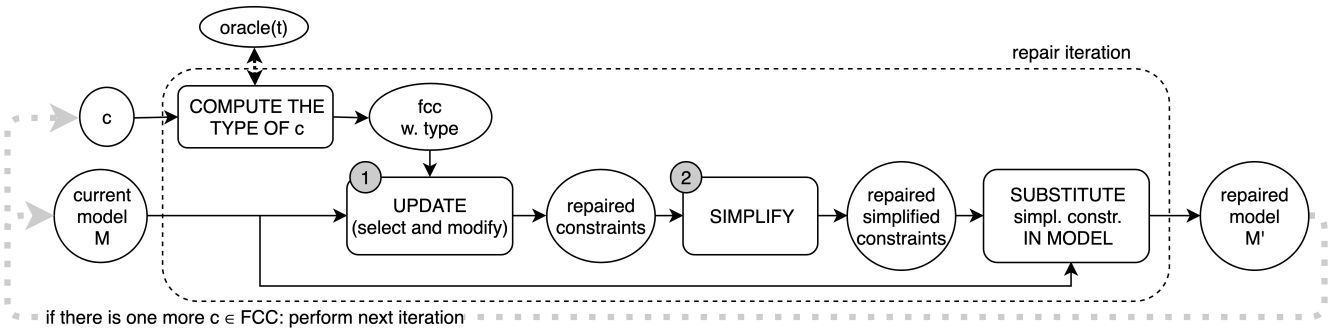


Figure 4: Single iteration of the optimized repair approach

3.2 Optimized repair approach

The naïve repair approach described in Sect. 3.1 could generate some redundancy, as shown in Ex. 4. Therefore, we introduce techniques for constraint selection and simplification to reduce the potential redundancy generated by the naïve approach. The goal is to make fewer edits as possible to the model, since we assume that a model with fewer edits better preserves domain knowledge.

Fig. 4 shows the optimized repair approach. It consists of three phases: (1) selection of some constraints to modify, (2) modification of the selected constraints, and (3) simplification of the modified constraints. Note that the process can be iterative if we identified more than one *fcc* (as in the experiments in Sect. 4). In the following, we consider one iteration of the process.

3.2.1 Update phase: selection and modification. To preserve the domain knowledge embedded in the constraints, we want the process to make as few changes as possible to them. Namely, we would like that the constraints Γ are updated with the following qualities:

(1) possibly no more constraints are added;
(2) as many constraints as possible are preserved identical;
(3) some constraints can be removed.

Therefore, the process performs a pre-processing phase, which selects only the constraints $\Gamma_S \subseteq \Gamma$ containing some configurations in common with the *fcc* c, and then modifies them. This phase is specific to the type of repair:

Strengthening repair Only the constraints γ_i sharing at least one feature with the *fcc* c, are selected. Formally, $\Gamma_S = \{\gamma_i \in \Gamma | (features(\gamma_i) \cap features(c)) \neq \emptyset\}$, where *features* collects the features contained in a formula. Then, the modification phase updates only one constraint γ_s selected randomly from Γ_S, by conjuncting $\neg c$. The repaired constraints set becomes $\Gamma' = (\Gamma \setminus \{\gamma_s\}) \cup \{\gamma_s \wedge \neg c\}$.

Weakening repair Only the constraints γ_i that exclude at least one configuration contained in the *fcc* c are selected. Formally, $\Gamma_S = \{\gamma_i \in \Gamma | isSAT(\neg\gamma_i \wedge c) = true\}$, where *isSAT* tells whether a formula is satisfiable or not. Then, the modification phase updates all the constraints in Γ_S by disjuncting them with c. The repaired constraints set becomes $\Gamma' = (\Gamma \setminus \Gamma_s) \cup \bigcup_{\gamma_i \in \Gamma_s} \{\gamma_i \vee c\}$.

3.2.2 Simplification phase. The constraint simplification procedure aims at reducing redundancy in the repaired model, especially when failure-containing combinations involve many features. A straightforward way to simplify a formula (or make it more readable) is to find the smallest, but equivalent expression. This problem is known as the *minimum-equivalent-expression* problem [14, 21].

We compared the three existing formula minimization techniques, and one minimization method based on mutations we have implemented (ATGT):

(1) JBool[1]: a tool that recursively applies logic rules and preprocessing techniques, preserving equivalence [13]: *literal removal*, *negation simplification*, *and/or reduplication and flattening*, *child expression simplification*, *and propagation*, *De Morgan's law*.
(2) Quine-McCluskey (QM) [27], a generalization of the Karnaugh Maps method. It requires the constraints to be in Disjunctive Normal Form (DNF) and its exponential complexity in the number of features makes it suitable only for small models (up to 15 features).
(3) Espresso[2], a faster version of the QM method that relies on some heuristics [37].
(4) ATGT[3] [15]: a hill-climbing process we implemented that iteratively mutates a formula randomly and checks it for equivalence.

We propose different methods, as we have seen that, in practice, these techniques often produce different outputs and none of them is guaranteed to generate an output which is always minimal compared to the others.

3.2.3 Correctness. Does the optimized approach produce correct repairs? A repair r is correct if it is equivalent to the one obtained by the naïve approach.

Theorem 2 (Correctness of the optimized approach). *The optimized approach is correct.*

Proof. The techniques applied in the *simplification phase* preserve equivalence. Therefore, we only show that the repairs computed in the *update phase* are equivalent to those computed by the naïve approach, because the two following properties hold:

- The strengthening repair is correct because of distributivity and commutativity of boolean conjunction: $\gamma_1 \wedge \ldots \wedge \gamma_s \wedge \ldots \wedge \gamma_m \wedge \neg c \equiv \gamma_1 \wedge \ldots \wedge (\gamma_s \wedge \neg c) \wedge \ldots \wedge \gamma_m$.

[1] JBool: https://github.com/bpodgursky/jbool_expressions
[2] Espresso logic minimizer, sources available at https://ptolemy.berkeley.edu/projects/embedded/pubs/downloads/espresso/index.htm
[3] ATGT: ASM Test Generation Tool. http://fmse.di.unimi.it/atgtBoolean.html

- The weakening repair is by definition equivalent to the naïve approach for all the constraints γ_i that are selected in Γ_S to be modified in the optimised approach (i.e., it performs the same operation of the naïve approach). It is correct also for each non-selected constraints γ_j, since for these constraints it holds $\gamma_j \vee c = \gamma_j$ (as c is *false* in the non-selected constraints): thus, γ_j can be left as it is. In fact, by translating this expression to a satisfiability problem, we obtain the condition under which the process does not select the constraint:

$$\begin{aligned}((\gamma_j \vee c) = \gamma_j) &\Leftrightarrow \neg isSAT((\gamma_j \vee c) \neq \gamma_j)\\ &\Leftrightarrow \neg isSAT((\gamma_j \vee c) \oplus \gamma_j)\\ &\Leftrightarrow \neg isSAT((\gamma_j \wedge \neg\gamma_j) \vee (c \wedge \neg\gamma_j) \vee (\neg\gamma_j \wedge \neg c \wedge \gamma_j))\\ &\Leftrightarrow \neg isSAT(\neg\gamma_j \wedge c)\end{aligned}$$

□

4 EVALUATION

In order to apply our process, we need a faulty variability model $\mathcal{M}$, a set of failure-containing combinations *FCC*, and an *oracle*. For the sake of experiments, we take as oracle another variability model $\mathcal{M}_o$, instead of the real oracle; in this way, we can also extract the set *FCC* by comparing $\mathcal{M}$ and $\mathcal{M}_o$.

4.1 Benchmarks

We have built two sets of benchmarks: $\text{BENCH}_{\text{MUT}}$ with seeded faults, and $\text{BENCH}_{\text{REAL}}$ with versioned models.

$\text{BENCH}_{\text{MUT}}$ *(seeded faults).* In order to build this benchmark set, we first selected some models to be used as $\mathcal{M}_o$, from previous papers and feature model repositories:

- example, from Example 1.
- register, a VSpec model for a register typically found in supermarkets, inspired by [38].
- django, an open source web application framework written in Python. Each Django project has a configuration file loaded at launch time. We considered 12 Boolean parameters (features), with constraints devised in our previous work [18].
- tight_vnc from FeatureIDE repository [28].

In order to obtain the initial faulty model $\mathcal{M}$, we seeded random faults in $\mathcal{M}_o$ using the following mutation operators:

- **RC**: removal of a constraint. There are studies showing that this is the most common case in practice [26].
- **RL**: removal of a literal in a constraint.
- **SL**: substitution of a literal in a constraint.

We generated 30 faulty versions $\mathcal{M}$ of each model $\mathcal{M}_o$ (10 with each mutation operator).

$\text{BENCH}_{\text{REAL}}$ *(versioned models).* For this benchmark set, we have considered two versions of variability models of the same system. We use the second version as oracle $\mathcal{M}_o$, and the first one as the faulty model $\mathcal{M}$. We picked three models of industrial applications from the SPLOT repository[4] [29]:

- the process model rhiscom, between versions 2.0 and 3.0;
- an enterprise resource planner (ERP-SPL);
- a windows accessibility module, between versions 7.0 and 8.0.

[4] http://52.32.1.180:8080/SPLOT/feature_model_repository.html

Table 2: Benchmarks size

	Name	# features	# constraints avg(min - max)	# literals
$\text{BENCH}_{\text{MUT}}$	example	3	1.67 (1-2)	3.0 (2-4)
	register	3	1.67 (1-2)	3.87 (2-5)
	django	12	4.6 (4-5)	10.87 (9-12)
	tight_vnc	24	11.67 (11-12)	53.2 (45-55)
$\text{BENCH}_{\text{REAL}}$	rhiscom	36	70	140
	ERP-SPL	43	75	151
	windows	335	943	2031

Table 2 reports the size of all the faulty models $\mathcal{M}$ to be repaired in the two benchmarks, in terms of number of features, number of constraints, and total number of literals in the constraints. For the constraints and literals of $\text{BENCH}_{\text{MUT}}$, it reports the average number across the 30 mutants and the minimum and maximum number between parentheses (the number of features is the same across the mutants).

4.2 Failure-containing combinations

For the sake of experiments, we obtain the set *FCC* from the faulty model $\mathcal{M}$ (having constraints Γ) and the model we use as oracle $\mathcal{M}_o$ (having constraints Γ_o), using the following process:

(1) first, we generate a test showing the difference (i.e., conformance fault) between the two models. The test is built as $t = getModel(\Gamma \neq \Gamma_o)$, where *getModel* returns a model of the propositional expression, if it exists, or *null* (in this case, the models are equivalent).
(2) then, we start from $c \leftarrow t$, and,
   - if $\Gamma_o(t)$, for each feature $f \in F$, if $\neg isSAT(\neg\Gamma_o \wedge rem(f, c))$ holds, then we do $c \leftarrow rem(f, c)$, where *rem* removes the assignment of f in c and returns the modified c.
   - if $\neg\Gamma_o(t)$, for each feature $f \in F$, if $\neg isSAT(\Gamma_o \wedge rem(f, c))$ holds, then we do $c \leftarrow rem(f, c)$.

This way we can obtain *fccs* that are as minimal as possible, and complete (i.e., the oracle is always true in case of over-constraining *fcc*, and always false in case of an under-constraining *fcc*).

4.3 Repair quality metrics

We want to assess the quality of a repair w.r.t. two goals: (i) simplification of the constraints, and (ii) minimization of the impact of edits. To this aim, we introduce two quality metrics that are used to compare Boolean expressions. We apply them to compare the conjunction of the constraints Γ of the original model $\mathcal{M}$ and the constraints Γ' of the repaired model $\mathcal{M}'$ obtained as output of the approach. The metrics are defined as follows:

- Complexity Distance (CD) as difference of formula sizes $CD(\Gamma, \Gamma') = literals(\Gamma) - literals(\Gamma')$, where *literals* returns the number of literals in a formula. As in [43], we also considered other measures (number of operators and node count

Table 3: Experimental results (mut.: mutation type; s.: strengthening repairs; w.: weakening repairs; ED: edit distance; CD: complexity distance; t: time in milliseconds, T/O: timeout occurred). In gray the best results (CD and ED over all the approaches, time over the simplification approaches)

			fccs and repairs		Naïve			onlySelection			simplification											
											ATGT			Espresso			JBool			QM		
	name	mut.	# (s.+w.)	size (s./w.)	CD	ED	t	CD	ED	t	CD	ED	t	CD	ED	t	CD	ED	t	CD	ED	t
BENCH$_{MUT}$	example	RC	1.0+0.0	2.0 / –	2.0	5.0	0.1	2.0	5.0	0.4	2.0	5.0	1064	2.0	5.0	53.4	2.0	4.0	5.3	2.0	4.0	24.2
		RL	0.3+1.0	2.0 / 1.0	3.8	10.8	0.2	2.2	6.0	0.3	2.2	6.0	1371	2.2	6.0	68.2	1.6	4.2	1.0	1.6	4.2	30.9
		SL	0.5+0.3	2.0 / 1.0	1.2	3.2	0.0	1.0	2.6	0.0	1.0	2.6	1060	1.0	3.0	52.4	1.0	2.6	1.1	1.0	2.6	23.3
	register	RC	1.0+0.0	2.6 / –	2.3	5.6	0.0	2.3	5.6	0.3	2.3	5.6	1065	2.3	5.6	52.4	2.3	4.9	1.0	2.3	4.9	24.0
		RL	0.2+1.1	2.6 / 1.3	5.5	14.6	0.0	2.9	7.7	0.3	2.5	6.9	1388	2.9	8.5	68.1	2.2	6.6	0.6	2.2	6.6	30.8
		SL	0.9+0.3	2.1 / 1.4	2.9	7.4	0.2	2.1	5.2	0.2	2.1	5.2	1433	2.1	6.0	69.2	2.1	5.7	1.2	2.1	5.7	32.9
	django	RC	0.5+0.0	1.7 / –	0.8	2.2	0.0	0.8	2.2	0.0	0.8	2.2	1062	0.8	2.2	52.2	0.8	1.3	1.0	0.8	1.3	24.2
		RL	0.4+1.4	2.0 / 4.0	8.0	22.8	0.0	1.6	4.4	0.6	1.2	3.2	2424	1.6	4.4	119.9	1.2	3.0	2.5	1.2	3.2	58.1
		SL	0.8+2.3	1.0 / 4.0	33.3	94.4	0.0	6.5	18.3	2.1	5.8	16.6	3720	6.5	19.2	180.3	7.4	21.7	4.2	5.8	18.4	116.0
	tight_vnc	RC	4.7+0.0	2.7 / –	14.0	39.1	0.1	14.0	39.1	11.0	14.0	39.1	8252	14.0	39.1	267	14.0	27.1	23.5	14.0	39.6	11199
		RL	0.7+31.8	1.9 / 14.2	2422	6000	1.2	202.2	499.5	402	–	–	T/O	202.2	499.5	2275	–	–	T/O	–	–	T/O
		SL	1.5+18.5	4.1 / 11.2	3734	8860	0.7	244.0	585.8	165	–	–	T/O	244.0	585.8	1369	–	–	T/O	–	–	T/O
BENCH$_{REAL}$	rhiscom	–	9+6	1.2 / 35.3	13977	30634	2	197	504	128	–	–	T/O	197	511	2717	–	–	T/O	–	–	T/O
	ERP-SPL	–	9+232	2.0 / 37.1	723426	1604116	15	16562	37273	8555	–	–	T/O	16562	37273	16562	–	–	T/O	–	–	T/O
	windows	–	989+55	2.3 / 453.8	8537492	17028426	87	175380	1918927	114028	–	–	T/O	174200	1917875	245532	–	–	T/O	–	–	T/O

in the parsed tree representation), but they do not change the overall results, therefore we do not report them here.

- Edit Distance (ED) computed between the syntactic trees of the two formulas Γ and Γ'. $ED(\Gamma, \Gamma')$ is defined as the number of *edits* (addition, substitution, or elimination) that we have to apply to Γ in order to obtain Γ'. A node of the tree can either be a literal or an operator. We use APTED as a tool to efficiently compute tree edit distances [32].

4.4 Experiments

We run experiments on the two benchmark sets BENCH$_{MUT}$ and BENCH$_{REAL}$: namely, we applied the naïve approach (see Sect. 3.1), the optimized approach (see Sect. 3.2) without the simplification phase (onlySelection), and with the simplification phase (employing the ATGT, Espresso, JBool and QM methods). Experiment code was written in Java and experiments were executed on a Linux PC with Intel(R) i7-3770 CPU (3.4 GHz) and 16 GB of RAM. All reported results are the average of 10 runs with a timeout for a single repair of 1 hour. The code and the benchmarks are available at https://github.com/fmselab/VMConstraintsRepair.

Results of the experiments are reported in Table 3. For benchmarks BENCH$_{MUT}$, results are categorized by the type of mutation. For each benchmark model, the table reports the number and size of strengthening and weakening repairs (note that each repair corresponds to one *fcc*); moreover, for each process setting, it reports the execution time, and the quality of the final model $\mathcal{M}'$ in terms of *CD* and *ED* distances. Values of the strategies ATGT, JBool and QM for repairing RL and SL mutations of tight_vnc and for all the benchmarks of BENCH$_{REAL}$ are not reported, because the experiment exceeded the timeout (T/O) of 1 hour.

We evaluate the process using three research questions.

RQ1: *Which quality do the constraints repaired by the process have?*

The main goal of this repair process is to not destroy domain knowledge. We consider the quality measures ED and CD to be proxies for domain knowledge preservation, under the assumption that having fewer edits means more preservation of the domain knowledge contained in the constraints. We therefore consider an approach *better* than another approach if it has smaller values of the quality measures.

The process (in all its versions) completely repairs all the benchmarks models, as the *fccs* in *FCC* are complete (see Thms. 1 and 2). However, the quality of the repaired models depends on the adopted repair approach. The optimized approach only using selection (onlySelection) always outperforms the naïve process in terms of quality of the repairs, as it modifies a subset of the constraints, and so the two measures CD and ED for it are always lower. The simplification approaches sometimes allow to obtain better repairs than onlySelection, meaning that they remove some redundancy introduced by the repair; however, there is no simplification method that is always better than the others on all the benchmarks for both measures (except for ATGT that is never worse than Espresso). We observe that, in a few cases, CD and ED are higher for a simplification method w.r.t. onlySelection (e.g., ED of Espresso for register RL): we have checked the example and we found that the simplification has removed some redundancy that was already present in the original model $\mathcal{M}$ so modifying the model more than what done by onlySelection.

RQ2: *How efficient is the repair approach?*

Computational time varies significantly for the different approaches and models: from 0-0.1ms of the smallest models, up to 245 seconds for the windows model (the biggest model having 335 features and 943 constraints) repaired with the Espresso simplifier.

The naïve approach, and the optimized approach without simplification (onlySelection), have been the fastest approaches; execution times of onlySelection are higher than the naïve approach for big

Table 4: Detailed results of the execution time for BENCH$_{REAL}$

	avg. repair time per single repair (ms)					
	selection			simplification		
name	str.	wea.	avg.	str.	wea.	avg.
rhiscom	2.6	4.7	3.4	57	54.3	55.9
ERP-SPL	4.3	17.7	17.2	119.8	123.3	123.2
windows	49.5	697.2	83.6	106.9	104.6	106.8

models, as it uses a SAT solver for identifying the constraints that must be repaired (see Sect. 3.2.1). Simplification algorithms are the main responsible for slow performance in terms of computation time. ATGT is the slowest one, as it internally calls a SAT solver several times, and the algorithm is yet in a prototypical stage. The second slowest simplification method is Espresso, but we noticed that it is relatively faster than other methods for large models; indeed, it is the only approach able to simplify all the benchmark models, while the others cannot simplify the biggest mutations obtained for tight_vnc and all the models in BENCH$_{REAL}$ in the given timeout. For small models, JBool is the fastest simplification method, followed by QM; however, they both timeout for large models.

RQ3: *Is there a repair type that our process handles more efficiently?* We are here interested in investigating whether there is an effect of the type of repair on the performances of the process. In order to better understand the computational cost of the type of repairs, Table 4 reports, for BENCH$_{REAL}$, the average execution times of strengthening and weakening repairs in the selection and simplification phases, and also the average time of any repair (regardless of the type). We only report the results of Espresso, as it is the only tool that completes before the timeout of 1 hour.

We observe that, in the selection phase, strengthening repairs are faster than weakening repairs: indeed, the former ones only do a syntactical analysis of the constraint, while the latter ones need to call a SAT solver (see Sect. 3.2.1). The average repair time in the selection phase is then influenced by the number of repairs of the two types: in ERP-SPL, since almost all the repairs are weakening (see Table 3), the average time is mostly influenced by them; in windows, instead, most of the repairs are strengthening (see Table 3) and so the average repair time is influenced by them.

Regarding the simplification phase, there is no significant difference between the two types of repairs.

5 THREATS TO VALIDITY

We discuss the threats to the validity of our results along two dimensions.

External Validity. Regarding external validity, a first threat comes from the choice of the variability models on which we performed the experiments. In the benchmarks, we totally selected models of seven applications of different sizes, among them two industrial applications from the SPLOT public repository. Although we have not tested our process on bigger feature models, we believe that the number and variety of input data make the results of our evaluation generalizable to other models of similar size.

In BENCH$_{REAL}$, we *simulated* real faults in constraints by enumerating the failure-containing combinations (and thus the single repairs) between two versions of constraints. Such simulated faults may not be accurate with respect to real usages in some scenarios. However, we believe that such results may be generalizable in cases when the faults are automatically detected by testing the *updated* system implementation, with respect to an *outdated* model, as we believe that the second version of the model accurately reflects the underlying system implementation.

Internal Validity. Regarding internal validity, a first threat involves the number of experiments and the accuracy of results. To this aim, we executed the experiments 10 times.

Another threat comes from the metrics used to assess the effectiveness and efficiency of our approach, not being a good proxy for domain knowledge preservation. We believe, however, that the chosen metrics well represent the concepts of formula readability and impact of the changes (ED), that may be useful for successive reasoning, for a reverse engineering process from propositional formula to feature model, and for comprehension by the user.

Regarding our approach in general, we have identified the following two threats to validity. The first one regards the applicability of our repair technique. We assume that the variability model is given as a set of constraints, while in general other formats (like feature models) are widely used. However, it is almost always possible to extract the set of features and the constraints among them, so our approach is generally applicable. It is true that it may be not easy to go back from the repaired model to the original format (see Fig. 3), but we try to change the model as little as possible. This should ease the identification of the applied repairs and facilitate the reverse process to extract the final variability model in another format.

The second threat regards the assumption of the *fccs* completeness. In general, we may find some conformance faults, but it may be not enough, since our process assumes that the *fccs* are complete. However, we can notice that every failing test is a complete *fcc* regardless of the test suite. Trying to extract a smaller failing combination from a failing test possibly requires new tests, but there already exist several techniques for fault localization that efficiently can do that [20].

6 RELATED WORK

There exist methods to statistically infer constraints from sampled configurations [1, 3, 16, 39–41]: they use a classifier to infer the conditions among parameter values, that determine a particular property, either a parameter above/below a certain threshold (like in [40]), or directly the configuration being accepted or rejected by the system (as in [41]). These machine-learning based methods are well-documented and supported by application studies to real scenarios, such as learning constraints among parameters in SCAD programs that may cause defects of configurable objects to 3D print [3]; and, in the case of LaTeX, showing that it is possible to obtain constraints among Boolean or numerical values, to format the paper to meet desired properties, such as a defined page limit [2]. Another interesting application of inferring constraints using machine learning is the case of mining temporal and value constraints from rich logs, for event-based monitoring in industrial SoS (Systems of Systems) [24]. The approach, integrated also with

techniques from process mining and specification mining, was applied to the automation system of a metallurgical company, and it consists of a *ranking* phase, based on some validity ratio metrics on the classification tree outcome, in which constraints that are more likely to be accepted by the users appear first.

The constraints learned (or *mined*) with such machine learning methods achieve a good accuracy (greater than 80% on average [40]), and they are able to completely *infer* constraints from scratch, or to *specialize* the model by adding the new inferred constraints to the existing ones. Our approach, however, is focused on performing any kind of repair to an existing set of constraints, and is also able to *generalize* the model, or apply *arbitrary edits* (as described in the classification in [42]). Moreover, our proposed method focuses only on the *manipulation* of existing constraints, and not on the actual detection of the failure-containing combinations, that we assume as input of our process. Unlike our approach, that takes the failure-containing combinations as input, those ML processes also include automatic detection of such *fccs* (in the form of constraints), given a sample of configurations classified as valid or non-valid [40]. For these reasons, the approaches are not alternative but complementary, as our approach is not comparable to those ML methods; however, as future work, we believe that it could be interesting to combine our process with those machine learning approaches, to have a more complete process for real case scenarios.

A quality-based model refactoring framework assessing the quality of merging operations among SPL models, expressed in UML [36], supports maintainability of models describing relations among features. It represents another approach to model repair, although it is not focused on repairing constraints in propositional logic. There is a comprehensive general work on repair of models by Reder et al. [35], with a method to detect inconsistencies using a validator, and to generate a repair tree representing in a compact way all the different viable actions to repair the model. This approach has been evaluated on UML models and OCL design rules, and is currently integrated in the Model/Analyzer plug-in for the IBM Rational Software Architect (RSA). We believe that our approach, instead, is a particular case of such repair framework in which the repair actions are fixed and determined by the selection and simplification algorithms (i.e., Espresso, QM, etc.), whereas the inconsistency detection is left to the engineer, who has to provide a set of failure-containing combinations in input to our process. However, despite our process has a fixed repair type and in this paper we evaluated its application, it may be possible to integrate the idea of [35] and build a sort of repair-action tree in which the *fccs* are applied in different order, for example, or with a different simplification method for each *fcc*, and we believe that the result of following another path in that repair-action tree could give slightly different results (that could be better or could also be worse).

Program repair techniques, such as SemFix [31], GenProg [25], and Par [23] already apply successive patch transformations, but to repair single faults in the code directly.

A process to detect and repair feature models from *conformance faults* with respect to another model has been presented in [11]. Our work, however, is able to handle arbitrary constraints of a variability model, and adds also the simplification of such modified constraints. The need for a fault-driven constraint repair process was already envisioned in [22], but no experiments were yet performed.

In the classification of edits to variability models presented in [26], our process fits the categories *build fix* and *adherence to changes in code*; in the classification of edits to variability models presented in [42], our process is able to address all kind of edits: in the case of *arbitrary edits*, it achieves them by applying *specialization* (what we call *strengthening repair*) and *generalization* (what we call *weakening repair*) sequentially.

A different technique for feature model repair in the context of system evolution, with different versions of systems, used mutation operators to make the model meet a specific *update request* [6, 7]; however, that approach does not handle arbitrary constraints, and does not guarantee to completely fulfil the update request, and thus to repair the model. We believe that that approach could be extended with our process, to be able to *repair* not only the feature *tree*, but the constraints as well.

Repair of constraints has usage also in other contexts, such as the repair of parameter values of timed automata clock guards, by applying tests and specializing the constraints [5]; and in the detection of constraints among parameters that let the built attack string trigger an XSS vulnerability in the system [19].

7 CONCLUSION

We proposed a process that, given a (faulty or outdated) variability model, and the faults in terms of failure-containing combinations, identifies the constraints involved in the fault, repairs them according to the oracle value, and simplifies them to make the edit minimal. We conducted an empirical evaluation on 7 models of different sizes, and found that the process of selecting only some constraints is indeed more effective than the naïve approach that modifies all of them. Moreover, we observed that simplification approaches can further improve the quality of the repair. However, their applicability is limited by the model size, as most of them do not scale on big models.

As future work, we plan to adapt our approach to larger models and to include in the evaluation the performances of reverse engineering the final constraints into a variability model (in case the repairs affected the structure of the initial variability model). As future work, we also want to address the current limitations; for example, by designing better selection and simplification strategies, by extending the method to non-boolean variables, and by including new simplification techniques. Furthermore, in order to better preserve domain knowledge, we plan to design an approach that interacts with domain engineers, for instance by highlighting implicit constraints as in [4].

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] Hadil Abukwaik, Mohammed Abujayyab, Shah Rukh Humayoun, and Dieter Rombach. 2016. Extracting conceptual interoperability constraints from API documentation using machine learning. ACM Press, 701–703. https://doi.org/10.1145/2889160.2892642 00002.

[2] Mathieu Acher, Paul Temple, Jean-Marc Jézéquel, José A. Galindo, Jabier Martinez, and Tewfik Ziadi. 2018. VaryLATEX: Learning Paper Variants That Meet

Constraints. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 2018)*. ACM, New York, NY, USA, 83–88. https://doi.org/10.1145/3168365.3168372

[3] Benoit Amand, Maxime Cordy, Patrick Heymans, Mathieu Acher, Paul Temple, and Jean-Marc Jézéquel. 2019. Towards Learning-Aided Configuration in 3D Printing: Feasibility Study and Application to Defect Prediction. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS '19)*. ACM, New York, NY, USA, Article 7, 9 pages. https://doi.org/10.1145/3302333.3302338

[4] Sofia Ananieva, Matthias Kowal, Thomas Thüm, and Ina Schaefer. 2016. Implicit Constraints in Partial Feature Models. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development (FOSD 2016)*. ACM, New York, NY, USA, 18–27. https://doi.org/10.1145/3001867.3001870

[5] Étienne André, Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. 2019. Repairing Timed Automata Clock Guards through Abstraction and Testing. In *Proceedings of 13th International Conference on Tests and Proofs (TAP 2019)*. Springer International Publishing. (to appear).

[6] Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. 2018. An Evolutionary Process for Product-driven Updates of Feature Models. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 2018)*. ACM, New York, NY, USA, 67–74. https://doi.org/10.1145/3168365.3168374

[7] 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

[8] Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. 2019. Efficient and Guaranteed Detection of t-Way Failure-Inducing Combinations. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 200–209. https://doi.org/10.1109/ICSTW.2019.00054

[9] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. 2015. How to Optimize the Use of SAT and SMT Solvers for Test Generation of Boolean Expressions. *Comput. J.* 58, 11 (2015), 2900–2920. https://doi.org/10.1093/comjnl/bxv001

[10] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. 2015. Generating Tests for Detecting Faults in Feature Models. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. 1–10. https://doi.org/10.1109/ICST.2015.7102591

[11] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. 2016. Automatic Detection and Removal of Conformance Faults in Feature Models. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 102–112. https://doi.org/10.1109/ICST.2016.10

[12] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Conference on Software Product Lines (SPLC'05)*. Springer-Verlag, Berlin, Heidelberg, 7–20. https://doi.org/10.1007/11554844_3

[13] Armin Biere. 2012. Preprocessing and Inprocessing Techniques in SAT. In *Hardware and Software: Verification and Testing*, Kerstin Eder, João Lourenço, and Onn Shehory (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–1.

[14] David Buchfuhrer and Christopher Umans. 2011. The Complexity of Boolean Formula Minimization. *J. Comput. Syst. Sci.* 77, 1 (Jan. 2011), 142–153. https://doi.org/10.1016/j.jcss.2010.06.011

[15] Andrea Calvagna and Angelo Gargantini. 2009. Combining Satisfiability Solving and Heuristics to Constrained Combinatorial Interaction Testing. In *TAP (Lecture Notes in Computer Science)*, Catherine Dubois (Ed.), Vol. 5668. Springer, 27–42. http://dx.doi.org/10.1007/978-3-642-02949-3

[16] Fei Chiang and Renee J. Miller. 2011. A unified model for data and constraint repair. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 446–457. 00046.

[17] John Joseph Chilenski and Steven P. Miller. 1994. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal* 9 (September 1994), 193–200(7). Issue 5.

[18] Angelo Gargantini, Justyna Petke, and Marco Radavelli. 2017. Combinatorial Interaction Testing for Automated Constraint Repair. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 239–248. https://doi.org/10.1109/ICSTW.2017.44

[19] Bernhard Garn, Marco Radavelli, Angelo Gargantini, Manuel Leithner, and Dimitris E. Simos. 2019. A Fault-Driven Combinatorial Process for Model Evolution in XSS Vulnerability Detection. In *Advances and Trends in Artificial Intelligence. From Theory to Practice*, Franz Wotawa, Gerhard Friedrich, Ingo Pill, Roxane Koitz-Hristov, and Moonis Ali (Eds.). Springer International Publishing, Cham, 207–215.

[20] Laleh Sh Ghandehari, Jaganmohan Chandrasekaran, Yu Lei, Raghu Kacker, and D. Richard Kuhn. 2015. BEN: A combinatorial testing-based fault localization tool. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 1–4.

[21] Edith Hemaspaandra and Henning Schnoor. 2011. Minimization for Generalized Boolean Formulas. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume One (IJCAI'11)*. AAAI Press, 566–571. https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-102

[22] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. Towards automated testing and fixing of re-engineered feature models. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 1245–1248.

[23] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, San Francisco, CA, USA, 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[24] Thomas Krismayer, Rick Rabiser, and Paul GrUnbacher. 2017. Mining constraints for event-based monitoring in systems of systems. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Urbana, IL, 826–831. https://doi.org/10.1109/ASE.2017.8115693

[25] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 3–13.

[26] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the linux kernel variability model. *Software Product Lines: Going Beyond* (2010), 136–150.

[27] E. J. McCluskey. 1956. Minimization of Boolean Functions*. *Bell System Technical Journal* 35, 6 (1956), 1417–1444. https://doi.org/10.1002/j.1538-7305.1956.tb03835.x

[28] 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

[29] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. SPLOT: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 761–762.

[30] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 140–151. https://doi.org/10.1145/2568225.2568283

[31] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 772–781.

[32] Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. *Information Systems* 56 (March 2016), 157–173. https://doi.org/10.1016/j.is.2015.08.004

[33] Justyna Petke, Myra B. Cohen, Mark Harman, and Shin Yoo. 2015. Practical Combinatorial Interaction Testing: Empirical Findings on Efficiency and Early Fault Detection. *IEEE Trans. Software Eng.* 41, 9 (2015), 901–924. https://doi.org/10.1109/TSE.2015.2421279

[34] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin, Heidelberg.

[35] Alexander Reder and Alexander Egyed. 2012. Computing repair trees for resolving inconsistencies in design models. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*. ACM Press, Essen, Germany, 220. https://doi.org/10.1145/2351676.2351707

[36] Julia Rubin and Marsha Chechik. 2013. Quality of merge-refactorings for product lines. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 83–98.

[37] Richard L. Rudell. 1986. *Multiple-Valued Logic Minimization for PLA Synthesis*. Technical Report UCB/ERL M86/65. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/1986/734.html

[38] Daisuke Shimbara and Øystein Haugen. 2015. *Generating Configurations for System Testing with Common Variability Language*. Springer International Publishing, Cham, 221–237. https://doi.org/10.1007/978-3-319-24912-4_16

[39] Paul Temple, Mathieu Acher, Battista Biggio, Jean-Marc Jézéquel, and Fabio Roli. 2018. Towards Adversarial Configurations for Software Product Lines. *CoRR* abs/1805.12021 (2018). arXiv:1805.12021 http://arxiv.org/abs/1805.12021

[40] Paul Temple, Mathieu Acher, Jean-Marc Jezequel, and Olivier Barais. 2017. Learning Contextual-Variability Models. *IEEE Software* 34, 6 (Nov. 2017), 64–70. https://doi.org/10.1109/MS.2017.4121211

[41] Paul Temple, José A. Galindo, Mathieu Acher, and Jean-Marc Jézéquel. 2016. Using Machine Learning to Infer Constraints for Product Lines. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC '16)*. ACM, New York, NY, USA, 209–218. https://doi.org/10.1145/2934466.2934472

[42] Thomas Thum, Don Batory, and Christian Kastner. 2009. Reasoning about edits to feature models. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 254–264. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5070526 00173.

[43] 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 Press, Piscataway, NJ, USA, 178–188.

Variability Management in a Software Product Line Unaware Company: Towards a Real Evaluation

Ana E. Chacón-Luna
Department of Engineering Sciences
University of Milagro
Milagro, Ecuador
achaconl1@unemi.edu.ec

Elvira G. Ruiz
José A. Galindo
David Benavides
Department of Languages and Computer Systems
University of Seville
Seville, Spain
{elvgarrui,jagalindo,benavides}@us.es

ABSTRACT

Software Product Lines (SPL) enable systematic reuse within an organization thus, enabling the reduction of costs, efforts, development time and the average number of defects per product. However, there is little empirical evidence of SPL adoption in the literature, which makes it difficult to strengthen or elaborate adjustments or improvements to SPL frameworks. In this article, we present the first steps towards an empirical evaluation by showing how companies that do not know about of SPL manage variability in their products, pointing out the strengths and weaknesses of their approaches. To this end, we present the design of a *case study* that we plan to carry out in the future in two companies to evaluate how companies perform variability management when they are not aware of software product lines. Our assumption is that most of the companies manage variability but no many of them are aware of software product lines. In addition, the first preliminary results of the case study applied in a company are presented.

CCS CONCEPTS

• **General and reference → Empirical studies**; **Design**; • **Software and its engineering → Software product lines**;

KEYWORDS

a case study, software product lines, variability management

ACM Reference Format:
Ana E. Chacón-Luna, Elvira G. Ruiz, José A. Galindo, and David Benavides. 2019. Variability Management in a Software Product Line Unaware Company: Towards a Real Evaluation. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, Article 7, 8 pages. https://doi.org/10.1145/3307630.3342421

1 INTRODUCTION

Nowadays, Software Product Lines (SPL) are being increasingly used by organizations as a means to achieve improvements in quality and productivity, as well as decreasing time-to-market and costs.

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

ACM ISBN 978-1-4503-6668-7/19/09.
https://doi.org/10.1145/3307630.3342421

According to Clements and Northrop [5], an SPL represents a set of software systems which share a common set of *features*, that satisfies the specific needs of a particular domain or market segment, and which is developed from a *core assets* common system in a pre-established way. The transition into a Software Product Line Engineering (SPLE) paradigm is not easy. In Ahnassay et al. [1], a systematic literature review of empirical evaluations in software product lines was carried out, the analyzed literature was collected between 2006 and 2011. After obtaining the information, the conclusion was that only 34% of these investigations involved industry professionals, and most of the articles did not provide any specific foundation for the problem statement but simply described the solution. They also revealed problems in several aspects of the quality assessment criteria for the research design and the report presentation. None of the studies found examples of companies which do not know about software product lines.

In Benavides et al.[4], a study was carried out on how companies which do not know about software product line concepts manage variability. It was concluded that the company under observation had some variability management practices, such as the reuse of product assets among other aspects. However, this study was carried out in a non-systematic way, without putting into practice a formal method of empirical evaluation, such as experiments or case studies. To the best of our knowledge, there are no existing empirical systematic studies on how variability management is handled by companies which do not explicitly know the methods, processes, tools, strategic methodologies and technologies of software product line. Furthermore, there is not an established known path for the transition of companies which want to implement SPLE approaches. For this reason, we consider that there is a research gap.

The contribution of this article focuses on the design of the case study applying the methodology described by Runeson et al. [13]; to determine how variability is managed by companies which do not know the software product line concepts. We also present the first results of the execution of the case study in Company A. We are based on the hypothesis that, there are more companies managing variability than companies which know about SPL. We will study the methodologies applied in companies. Later, we will contrast them with the activities and/or processes of the SPL framework defined by Pohl et al. [11]. Finally, we will design a case study for companies which somehow manage variability in their products. The objective of this study is to determine which SPL concepts are used in practice. As future work,the results of the case study

can let us define some adoption practices of transition to the SPL paradigm.

The rest of the article is structured as follows: Section 2 briefly describes the software product lines framework, as indicated in Pohl et al. [11]; Section 3 covers related studies. In Section 4, the case study design is developed specifying: the companies that will be involved, the analysis units to be studied, type of case study, research questions, data collection methods, data selection strategies, data analysis method according to the plan; in Section 5 presents the first preliminary results of the case study; in Section 6 we shown how the threats to validity were overcome in the case study execution; and Section 7 outlines the conclusions and future works.

2 SOFTWARE PRODUCT LINES

There are several frameworks related to the processes, concepts or activities used in SPL engineering. In this case, we start from a widely accepted differentiation for all the proposals between domain engineering and application engineering proposed by Weiss and Lai [16].

According to Pohl et al. [11], SPL engineering consists of the following two processes. Firstly, *Domain engineering*, whose objective is to produce the platform, including applications commonality and variability; It consists of five key sub-processes: product management, domain requirements engineering, domain design, domain development and domain tests. Secondly; *Application engineering*, which aims to achieve the highest possible reuse of domain assets when developing a product; It consists of four sub processes: application requirements engineering, application design, application implementation and application tests.

With the aim of carrying out a good analytical procedure and considering that the studied companies are unaware of the existence of SPL, this study excludes the processes related to design, development, and testing. The reason for this decision is because the activities of those stages demand greater knowledge of the SPL paradigm.

Therefore, the empirical study proposal focuses on three processes: product management, domain requirements engineering and application requirements engineering. These processes were chosen because they are the least strict concerning technology to be applied and because we understand that a company executes these processes implicitly or explicitly to manage variability. However, a similar empirical study could be performed in the other processes, but then the scope of the study would be too broad, and that is the reason we previously reduced the scope to these three processes in this work.

Product management deals with the management aspects of the SPL and, particularly, the market strategy. Its purpose is the management of the organization's product portfolio, using scope techniques to specify what is inside and outside the scope of the product line. As a result, a product map will be obtained. This map establishes the main common and variable characteristics of the future products, as well as a calendar with their expected release dates. SPL product management differs from individual systems product management due to the following:

- An expected benefit of SPL engineering is the generation of product variants at a lower cost by reusing a large part of the assets developed in the domain engineering.
- The products in the product portfolio are closely related since they are based on a common platform.
- Product management takes into account the evolution of market needs, technology, and limitations for future applications, trying to anticipate possible changes in the features or legal limitations.

The domain requirements engineering includes the activities to obtain and document the common and variable requirements of the product line. The input for this sub-process consists of the product roadmap. The aim is to determine the reusable requirements, based on the variability model of the product line. Therefore, the results will show the specification of the common and variable requirements for all the product line applications. Domain requirements engineering differs from requirements engineering for individual systems because:

- The requirements are analyzed to identify those which are common to all applications and those which are specific to particular applications. The analysis of the requirements is documented in variability models.
- Based on input from the product management, domain requirements engineering predicts changes in requirements, laws, standards, technological changes and market needs for future applications.

The application requirements engineering performs the necessary activities to develop the specifications for application requirements. The reuse of domain artifacts depends on the requirements of the application. Therefore, an essential concern for application requirements engineering is detecting increments between the application requirements and the available capacities of the platform.

The *inputs* of this sub process are the domain requirements and a road-map of the product containing the main features of the corresponding application. Also, there may be specific requirements (for example, from a client) for the particular application that was not observed during the domain requirements engineering. The output is the specification of requirements for the particular application. The application requirements engineering differs from the requirements engineering for individual systems due to the following reasons:

- Obtaining the requirements is based on the communication of the common parts available and the variability of the software product line, considering that most of the requirements are derived from the domain requirements.
- The convergence of the application requirements and the domain requirements must be detected, evaluated according to the required adaptation effort and adequately documented. If the required adaptation effort is known in advance, it is possible to make balanced decisions about the application requirements to reduce the effort and increase the reuse of domain artifacts.

3 RELATED WORK

In Benavides and Galindo [4], a study was performed on how companies which do not know about software product line concepts

manage variability. It was concluded that the company under observation had some variability management practices, such as the reuse of product assets among other aspects. This study was carried out in a non-systematic way, without putting into practice a systematic method of empirical evaluation, such as experiments or case studies.

Although there are no previous documented studies about SPL practices in companies that are still unaware of SPL existence, there are several studies that can be considered as a base for our work. In da Silva et al. [6] and Bastos et al. [2] the authors present a complete study about SPL adoption in small and medium companies in which they aim to justify the use of agile methods and the use of a multi-method approach respectively for the adoption SPL.

In da Silva et al. [6] to achieve their goals, they present a single-case study based on the scoping and requirements of software, given the fact that those disciplines are the ones which define SPL life-cycle. Our case study is also based on the first sub processes of SPL engineering, with two medium-sized companies instead of just one. This will enable us to compare, extract similarities on both companies' practices and reach more robust conclusions.

In Bastos et al. [2] it is concluded that significant findings are an important step to establish guidelines for SPL adoption. The execution of our case study aims to determine how variability is managed by companies which do not know the concepts of the software product lines and to verify which SPL concepts are used in practice. As in our study, Bastos also has as objective, to define some transition practices for the implementation of SPL techniques.

In Rabiser et al. [12] a comparative study of the trend of research in SPL in the industrial and academic field is presented. This study revealed that more than one third (34%) of academic research provides artificial/toy examples only and 27% of the papers do not present any evaluation. This means that 61% of academic research is not properly validated. Based on the conclusion presented in the work of Rabiser et al. [12] where it is indicated that 61% of the studies are not correctly validated, we intend to validate through a case study the management of variability in companies which do not know SPL.

4 DESIGN OF THE CASE STUDY

For the elaboration of the case study, we will rely on the steps described in Runeson et al. [13] and Wohlin et al. [17].

The first step is the study design, where the objectives of the case study are planned by and defined. The second step is the preparation for the data collection, where the procedures and protocols for data gathering are defined. The third step is the data collection, to gather the data. The fourth step, the analysis of the collected data, is the procedure of analysis of the data, which later allows obtaining the conclusions of the study. Finally, in the fifth step, the presentation of reports, the results and conclusions must be disseminated, showing sufficient evidence of the study.

4.1 Planned context of the empirical evaluation

In this subsection, some context is given to the companies to achieve a better understanding of what small and medium-sized enterprises (PYMES) are. Because we do not have any legal authority to give the names of the companies, we will give them a generic name like A and B. Each company is located in a different part of the world, allowing the study to be less biased by social traditions of an area. It is expected to observe many differences in their working procedures, which will make it more interesting to get common factors in both of them.

Company A is a private company in South America. Its mission is related to financial and commercial development systems. Among its departments, there is the information technology department, which is composed of the planning department, the development department and the support and customer service department. It has 18 software products developed.

Company B is a public company located in Europe. Its mission is to implement the government policies of different municipal corporations in the field of modernization, innovation and implementation of ICTs and the computerization of different services for the benefit of several town halls and the self-management of town halls and other local entities of the province. They have developed 35 software products.

4.2 Analysis units

As previously described, the study will be conducted in two companies. Our analysis units will be the product management, the domain requirements engineering, and the application requirements engineering obtained from the engineering framework of the SPL Pohl et al. [11] mentioned in Section 2.

4.3 Type of case study

The research method to be applied in our case study will be: multiple embedded, flexible and exploratory case studies according to what is defined by Yin [18].

Multiple embedded case studies, as explained in Runeson et al. [13]„ it consists of two case studies of two different companies, with three analysis units each. The two companies studied use their methodologies in software development. The three analysis units are product management, domain requirements engineering and application requirements engineering [13]. The objective of this research is to study how software companies with similar features manage variability in their product development.

Flexible, because new information can be entered during data collection that may be important or critical to the study. In this way, the study design could be updated [13].

Exploratory, since we are interested in understanding the methodologies that companies use to manage variability between their products, this will allow us to explore the nature and weaknesses associated with them within a specific context [10].

4.4 Research questions

The main objective of the study is to determine how the variability is managed by Companies which do Not Know about Software Product Line concepts *(CNKSPL)*. We consider CNKSPL those companies that manage a family of systems and that might have solutions for variability management even if they are not aware of it. The definition of the research questions has been based on the description of the processes described in Pohl et al. [11], considering the analysis units of the present study.

We pose the following research questions:

RQ1. How does CNKSPL manage variability at product management level? To answer this question, we wondered:

- **RQ1.1**. What kind of measures are taken to manage the different products?
  The objective of this question is to identify if the products in the product portfolio handled by the company are related, since they are based on a common platform. Also, it makes it possible to verify if potential changes in characteristics, legal limitations and norms are foreseen for the future application of the SPL. As a result, the answer to this question can serve as an input for the definition and establishment of product management practices, and for generating product variants at an efficient developmental cost and time.
- **RQ1.2**. How do the strategies align with the product definition?
  The objective is to identify if the strategies of the department are aligned with the product definition and the choice of new product ideas. As a result from this question, through feedback from the roadmap, feature suggestions on the products can be included.
- **RQ1.3**. How is the maintenance of existing products carried out?
  The objective is to verify what activities are carried out in order to preserve and improve existing products in the market.
- **RQ1.4**. How is the market introduction of new products resolved?
  This involves identifying the distribution channels used, and the supply of new products, as well as the announcement of new products to potential customers
- **RQ1.5**. How is the product control process carried out?
  We want to determine if follow-up and training of the product management process are carried out, observing the sales volume of each product obtained.
- **RQ1.6**. Is there a product list detailing the features of each product? If so, what is its purpose?
  The objective of this question is to verify if the main common and variable features of all applications of the existing product line are defined. We also want to know if there is a calendar for the delivery of applications to specific clients or for the release in the market. If it does not exist, it is expected to be revealed how they manage the products of the same line

RQ2. How do companies manage variability at the level of domain requirements engineering?

- **RQ2.1**. Which means are most frequently used to obtain the requirements of the product?
  It is necessary to list the documents, mechanisms or techniques used to obtain the requirements for the development of new software products, whether these are common, unique in their product range or variables
- **RQ2.2**. When it comes to documenting requirements for products, is this done through natural language or a model?
  The purpose of this question is to identify how companies document their analysis of requirements and to find common parts in previously developed software. It also aims to reveal how they manage this process to optimize the functionality and development of the new product.
- **RQ2.3**. What are the models of requirements used to evidence the obtaining and analysis of requirements?
  We want to know how they indicate the obtaining of requirements, either through function models, data or behavior analysis, product comparison matrix, requirements matrix or analysis of similarities based on priorities [11].
- **RQ2.4**. What are the secondary sources used to obtain and create the requirements of the products? What is the purpose of using secondary sources?
  The goal is to identify all the sources they use to obtain the common and variable requirements. Also, to investigate whether creating a domain requirement artifact aims to reduce variability.

RQ3. How do the companies manage the variability at the level of application requirements engineering?

- **RQ3.1**. Is there clear documentation that shows the common and variable requirements of each application?
  The objective is to document the requirements of each application in their product portfolio.
- **RQ3.2**. When developing a new application, the requirements of another product are reused?
  The objective is to identify whether they have as a rule the reuse of artifacts from domain requirements
- **RQ3.3**. What happens when an application needs specific requirements for its creation?
  The objective is to know the criteria for modifying existing artifacts or developing a new artifact including the particular features of the application.

As a summary, in Table 1 we present the research questions in column two, based on concepts from the SPLE framework, and in column three we present the propositions which give us a general view of the information that will be obtained through the execution of the study instruments.

4.5 Data collection methods

Four data collection methods were used in this study: document analysis, observation, focus groups, and interviews

Documentation analysis, The documentation analysis is a technique that focuses on the documentation generated by software engineers. All documents that seem relevant to the investigation and that relate to their methodologies for product development will be analyzed. Product road-maps, application delivery schedule, documents that show product features, models of orthogonal variability and matrix of application requirements will be the main focus of this analysis. We expect companies to use different organizational schemes and therefore different names and techniques for documents containing that information.

Observation, This method will be applied as in [15] using the techniques of thinking aloud and direct observations, 120 hours will be used to make the observations in each company. It was planned to carry out 120 hours of observation considering the research team available time. Nevertheless, this suggestion may be changed depending on the company needs or requests, since we

IDs	RQs	PROPOSITIONS
How does CNKSPL manage variability at the level of product management?		
RQ1.1	What kind of measures are taken to manage the different products?	**If** product management measures are taken **then** the potential changes in the features, legal limitations and norms for future applications will be shown in advance. **If** product scope techniques are used, **then** those products that are part of a line and those that are outside the line will be specified. **If** product scope methods are defined **then** Product portfolio scoping or domain scoping or asset scooping are applied.
RQ1.2	How do the strategies align with the product definition?	**If** a product portfolio exists **then** the features specifications of each product will be provided. **If** there is a product portfolio **then** its usefulness is positively valued.
RQ1.3	How is the maintenance of existing products carried out?	**If** a list of products or development artifacts exists **then** maintenance is optimized. **If** existing artifacts that have already been developed in previous projects are identified **then** time and money in the creation of artifacts will be saved.
RQ1.4	How is the market introduction of new products resolved?	**If** a product delivery schedule is defined **then** marketing is optimally managed.
RQ1.5	How is the product control process carried out?	**If** there is a follow-up control of the products **then** the volume of sales of each product obtained may be determined
RQ1.6	Is there a product list detailing the features of each product? If so, what is its purpose?	**If** there is a roadmap **then** it has an important value for the company because it contains the features of the product and a schedule for market introduction. **If** tthe characteristics of each product are specified **then** the commonality and variability of the products are analyzed. **If** commonality and variability of products are analyzed **then** time and resources for the creation of new products will be saved.
How does CNKSPL manage variability at the level of domain requirements engineering?		
RQ2.1	Which means are most frequently used to obtain the requirements of the product?	**If** there are communication methods to obtain requirements common or variable requirements for the different products of the family **then** it is easier to collect them for later analysis.
RQ2.2	When it comes to documenting requirements for products, is it documented through natural language or a model?	**If** conceptual models exist for the definition of requirements **then** its usefulness is positively valued.
RQ2.3	What are the models of requirements that are used to evidence the obtaining and analysis of requirements?	**If** they use a requirements matrix of the applications **then** the concordance analysis will be more accessible. **If** commonality and variability requirements artifacts are defined, **then** an efficient analysis based on priorities is made. **If** an orthogonal variability model is used **then** an efficient analysis of the variants between products is performed.
RQ2.4	What are secondary sources used to obtain and create the requirements of the products? What is the purpose of using secondary sources?	**If** requirements from different sources are obtained, **then** the domain requirements artifacts will be more effective **If** the analysis of the domain requirements artifacts tries to minimize the variability of the requirements, **then** the effort that is invested on the flexibility of the design will be lower
How does CNKSPL manage variability at the level of application requirements engineering?		
RQ3.1	Is there clear documentation that shows the common and variable requirements of each application?	**If** there is documentation of the application requirements **then** the application variability model can be obtained.
RQ3.2	When developing a new application, the requirements of other product are reused?	**If** the application requirements specifications are common to the existing ones, **then** reuse the existing domain artifacts.
RQ3.3	What happens when an application needs specific requirements for its creation?	**If** the requirements for the application cannot be met by reusing artifacts from domain requirements **then** requirements artifacts are created for a specific application. **If** artifacts of specific requirements are needed for an application **then** the variable artifacts will be adapted to meet the needs. **If** specific requirement artifacts are needed for an application **then** the delta analysis should be performed to decide whether the delta requirements should be made for the application or not

Table 1: Research questions

are aware of how difficult it is for them to make a non-profitable collaboration with research groups. Even so, the time spent on each company will be even, so that we get the same depth of knowledge in both. Database developers, business programmers, front-end programmers, architects, and requirement analysts will be observed. The objective is to observe the process involved when obtaining the requirement of a new product, the support of existing products, the introduction of new products in the market, the process of product control, the creation of domain artifacts and their reuse.

Focus group, will meet as recommended in Kontio et al. [7]. These meetings will be held no more than 30 minutes once a week, in which it will be discussed how they manage their product variability, what methodologies they use to derive product requirements and address issues such as challenges and lessons learned and their relation to the development of SPL. Six principal analysts were selected to participate in the focus group, as they are the ones who actively interact in the definition of requirements and the definition of the scope of their products.

Interviews,

face-to-face interviews will be conducted with product managers, company programmers, and requirements analysts

4.6 Data selection strategy

The basic conditions to address our problem, goal, and research questions are: any SME's that have a software development department in a specific domain and share products, and the company is involved with the study. In the companies under study, participants will be selected based on the convenience sampling method [17]. This selection considers different roles and profiles involved within the company, which will be relevant to the investigation. For research purposes, it is proposed to select roles that involve project managers, product managers, business programmers, front-end programmers, architects, and requirements analysts, as well as all the roles that exist in the company related to the stages to be studied. For the specific selection, it is planned to have previous meetings with the company's managers as well as study the organizational documentation to know which specific roles would be the most appropriate to be linked to the study.

4.7 Data analysis methods

This study will analyze the data collected qualitatively, based on Miles and Huberman [9]. For data analysis, the tool *Nvivo* will be used, due to its qualitative characteristics of data analysis [3].

After the data collection, we will encode the data, which means that parts of the text will be given a code that represents a particular area, based on the objectives of the study. For each code a memo will be attached, that is, comments and reflections from the investigators with some problems or descriptions that allow describing the analysis units. Then, the first set of hypotheses will be paired, identifying, for example, phrases that are similar in different parts of the material and patterns in the data.

We must consider that the process is executed iteratively and affect each other, it is not a simple sequence of steps. Therefore, in the activity where hypotheses are identified if more information is required, more data collection is carried out in the field.

4.8 Threats to validity

Research questions: The research questions defined in this study may not focus on the most important aspects related to the three sub processes studied.
Interview questions: The proposed set of interview questions may not be explicit and prevent obtaining the necessary information. To mitigate this threat, an interview will be held in several sessions.
Observations: The observation will be carried out observing the employees who perform the activities related to the administration of the products and the analysis of requirements, documentation and artifacts. However, the company may not show all the required documentation. To mitigate this threat, confidentiality agreements will be signed with the company.
Selection of participants: As the selection was based on convenience sampling, we believe that the most appropriate participants will be selected to provide the appropriate information. We believe that the threat against the results obtained is reduced because the investigation will be carried out in two companies. However, when applying the study in SME's it is possible that a participant has more than one role within the company.

5 PRELIMINARY RESULTS

In this section we present the first series of results obtained from the intervention in company A.

5.1 Context

We have carried out the case study in an Ecuadorian company dedicated to financial and commercial payment means development since 1999. The company currently has more than seventeen national and foreign clients. In the technology and development department, there are approximately twenty two employees; Around nine project leaders, four analysts, six developers and three product certifiers. Currently, they have two star products: one desktop solution and another in web environment. The products have features that allow managing the operation of a financial company through five modules (transactional switch, credit cards, debit cards, payment points and Batch process).

5.2 Purpose of the study

The purpose of this case study is to determine how company A handles variability considering that it unaware of software product lines, however, it manages a portfolio of similar products which share common requirements.

5.3 Type of case study and Analysis Units

As previously described, the study will be conducted in two companies. Our analysis units will be the product management, the domain requirements engineering, and the application requirements engineering obtained from the engineering framework of the SPL [11] mentioned in Section 2. The method applied in this study was multiple embedded, flexible and exploratory case study.

Embedded, since two areas of company A were analyzed, these units were related to the scope of the product and the requirements analysis unit. With the information collected in company A, the analysis units of the present study were analyzed.

Exploratory, since we are interested in understanding how companies manage variability in their products. For this purpose, four methods were used to gather data which allow knowing if company A performed tasks applying SPL approaches.
Flexible, because we adapt to the needs presented in the study when gathering important data for analysis.

5.4 Data collection procedures

This study used four methods of data collection: interviews, focus groups, documentation analysis and observation. It is worth mentioning that, in order not to affect the obtaining of data and, because the people who collaborated in this study did not know about the SPL paradigm, all the instruments used were generic, subjective to the methodology that the company A uses. that is, SPL issues were not mentioned in the questions. However, the data collected allowed us to find out how they manage variability in their products. The instrument used can be found at Zenodo [1]

The main source of information in this study was the **semi-structured** interviews we conducted with the selected professionals in the company. It is worth mentioning that some of them perform more than one role within the company. Interviews were conducted with the manager, two project leaders with three analysts and one certifier.

Each interview was conducted in two sessions each of 1 hour. All the interviews were recorded in audio files that were then transcribed and coded for analysis.

A **focus group** was also carried out following the recommendations of Kontio et al. [8]. The focus groups were formed by seven employees who performed analysts, developers and certifiers roles. There were three sessions each of 30 minutes on Thursday morning. An important factor to mention was the willingness of the group to agree on criteria and to show examples of how they managed their product portfolio, and how they obtained the requirements of their new and old clients. We codified all the answers to the open questions posed to the participants.

In the development and technology deparment, **documentation analysis** was carried out, it was related to the product management processes and requirements analysis. Among these documents, the following were reviewed:
Request for Information (RFI) this document is generated by the client, it contains: customer data and describes the current business scenario, it also details how they carry out the processes and finally describes the future scenario detailing what they need, that is, specifying the features needed for the product to comply.
Request for Proposal (RFP) this document is generated by the company and it details the features the product will have.
Agreement act between the clients and the company, this document details the information collection requested by the client, also in this document, it is found the delivery planning of the product through a Gantt chart which systematically details the resources and the time allocated for the execution of each activity within the project for the delivery of the product.
Another method used was: **observation**. This method followed the advice of Seaman Seaman [14]. Direct observations were made, field notes in place. The observed roles were: manager, analysts, certifiers. The time allocated to perform the observations to extract information on the first unit of analysis (product scope) was 40 hours and another 50 hours were allocated for the observation of the second unit of analysis, that is, the requirements analysis. Through this method, valuable information was gathered, such as the face-to-face interactions between clients and the company's personnel when obtaining and negotiating the requirements of a product. E-mail messaging was also observed in order to know the product requirements are collected.

It could be observed that the company has two basic products, the ProdN1 can be defined as a single product, but there are many ways to use it. The system is modular. It has five main modules and also other submodules. The implementation of these modules varies, some modules are central and are always implemented, whereas the use of the other modules depends on the requirements of each client.

company A clients vary in size and operation, the ProdN1 tool is built to be customizable. The business operation of the company A is based on managing a set of predefined configurations for the most common product requirements, but if the client requests a variant of the modules available, there is also a support option for the customization of more variants.

However, in some cases, a customer has needs beyond the functionality available on the ProdN1 and ProdN2 platform they maintain. In these cases, company A can provide a tailor-made solution as a last resort, developing the new requirement for the client even if it is out of reach.

5.5 key finding(s)

After analyzing the data collected from company A, the following key findings were obtained: **Product Management**

- The products of company A are related, since they are based on a star product or common platform.
- The products they develop are always within the scope of the company's mission. In other words, they only develop products related to financial and commercial development systems.
- The maintenance of existing products is carried out when the client requests it, that is, they do not anticipate modifications to their star product.
- They do not anticipate the creation of new products without the customer's requirement.
- There is a list of the products with the particular features or functions of each product. However, this list is purely informative. There is no matrix where the common or variable characteristics of each product are related. The objective of this list is to know the variants of the products to offer new customers products with similar features in order to save cost and development time.

Requirements domain engineering

- To obtain the requirements of their products, they do so through forms that detail the functions and features of each product through natural language. No comparisons are made with other products.
- One of the objectives of the star product or base platform is to minimize particular requirements for each product.

[1] https://doi.org/10.5281/zenodo.3261266

- There is no document that relates or compares the common and variable features among the products.

Requirements application engineering

- When developing a new application, the requirements of another product are reused.
- When a new client requests the development of a product, a large part of the star product is always reused, configuring or adapting the existing parts. Rarely, a new module was developed, always trying to configure to use the existing ones.

6 DISCUSSION

In the present study, the first preliminary results of the application and execution of the case study were presented. However, it must be emphasized that, this first approach to the industry allowed to improve the initial planning of the case study design. For example, we had to rethink the number of hours planned to make observations in the company. Even though we had the authorization and the commitment to collaborate with the company, 120 hours of observation seemed to be excessive for the company since they were dealing with business processes, therefore, observation time had to be shortened to 40 hours in order to observe information on topics related to the scope of the product and 50 hours to observe how they manage the requirements analysis.

Another rethinking of the original design of the case study was related to the execution of the first interview with an analyst, because it allowed us to realize that the instrument was not so explicit and avoided obtaining all the necessary information relevant to how they performed their products maintenance process. For the next session, modifications were made to this instrument so that the pertinent information could be obtained.

7 CONCLUSIONS

In this paper, we present the design of the study, defining the objectives and scope, and also identify the protocols and procedures to be followed for the collection of data. After the execution of the case study in the first company, it was concluded that company A has defined a portfolio of products that specifies each product features, additionally, the scope of its products is defined, it also has a common platform. However, they do not have a document that analyzes the common or variable parts of their product portfolio. In addition, they do not anticipate maintenance of their products either. We can infer that the variability at the level of product management in company A. is partially managed.

However, no positive results were obtained from variability management at domain requirements levels since there are no conceptual models for the definition of requirements, they do not perform an analysis of the product requirements, nor do they use orthogonal variability models. We infer that, company A does not comply with activities carried out in the domain requirements engineering. Since it does not manage variability at domain requirements engineering level.

Regarding Variability management at the engineering level of application requirements, the principle of reuse is respected, that is, if the specifications of the application requirements are common to those already existing, then what is already existing is reused.

As future work, we consider the execution of the case study in more companies, the objective is to analyze the joint results and determine the approaches or methodologies applied by the companies, we intend to contrast with the SPL concepts seeking to reveal how these companies manage variability. It will also be possible to determine to what extent companies are applying SPL approaches in industrial contexts. These results will define some adoption practices to make the transition to SPL, which will strengthen the SPL framework as a result of these studies.

8 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); University of Milagro; and the Junta de Andalucia METAMORFOSIS project.

REFERENCES

[1] Alvin Ahnassay, Ebrahim Bagheri, and Dragan Gasevic. 2013. *Empirical evaluation in software product line engineering.* Technical Report. Tech. Rep. TR-LS3-130084R4T, Laboratory for Systems, Software and Semantics, Ryerson University.

[2] Jonatas 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

[3] Patricia Bazeley and Kristi Jackson. 2013. *Qualitative data analysis with NVivo.* Sage Publications Limited.

[4] David Benavides and José A. Galindo. 2014. Variability management in an unaware software product line company: an experience report. In *The Eighth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '14, Sophia Antipolis, France, January 22-24, 2014.* 5:1–5:6. https://doi.org/10.1145/2556624.2556633

[5] Paul Clements and Linda Northrop. 2002. *Software product lines.* Addison-Wesley,.

[6] Ivonei Freitas da Silva, Paulo Anselmo da Mota Silveira Neto, Pádraig O'Leary, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. 2014. Software product line scoping and requirements engineering in a small and medium-sized enterprise: An industrial case study. *Journal of Systems and Software* 88 (2014), 189–206. https://doi.org/10.1016/j.jss.2013.10.040

[7] Jyrki Kontio, Johanna Bragge, and Laura Lehtola. 2008. The focus group method as an empirical tool in software engineering. In *Guide to advanced empirical software engineering.* Springer, 93–116.

[8] Jyrki Kontio, Johanna Bragge, and Laura Lehtola. 2008. The focus group method as an empirical tool in software engineering. In *Guide to advanced empirical software engineering.* Springer, 93–116.

[9] Matthew B Miles and A Michael Huberman. 1994. *Qualitative data analysis: An expanded sourcebook.* SAGE Publications, inc.

[10] Michael Quinn Patton. 1990. *Qualitative evaluation and research methods.* SAGE Publications, inc.

[11] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques.* Springer Science & Business Media.

[12] 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 *Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018.* 14–24. https://doi.org/10.1145/3233027.3233028

[13] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. 2012. Case study research in software engineering. In *Guidelines and examples.* Wiley Online Library.

[14] CB Seaman. 2008. Qualitative methods. Guide to advanced empirical software engineering: 35–62.

[15] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering* 25, 4 (1999), 557–572.

[16] David M. Weiss and Chi Tau Robert Lai. 1999. Software product-line engineering: a family-based software development process. (1999).

[17] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering.* Springer Science & Business Media.

[18] Robert K Yin. 2003. Investigación sobre estudio de casos. Diseño y métodos. *Applied Social Research Methods Series* 5 (2003).

Analyzing the Convenience of Adopting a Product Line Engineering Approach: An Industrial Qualitative Evaluation

Luisa Rincón
Dpto. de Electrónica y Ciencias de la Computación, Pontificia Universidad Javeriana - Cali
Cali, Colombia
Centre de Recherche en Informatique, Université Paris 1 Panthéon-Sorbonne
Paris, France
lfrincon@applies.variamos.com

Raúl Mazo
Centre de Recherche en Informatique, Université Paris 1 Panthéon-Sorbonne
Paris, France
GIDITIC, Universidad EAFIT
Medellin, Colombia
Lab-STICC, ENSTA Bretagne
Brest, France
raul.mazo@univ-paris1.fr

Camille Salinesi
Centre de Recherche en Informatique, Université Paris 1 Panthéon-Sorbonne
Paris, France
camille.salinesi@univ-paris1.fr

ABSTRACT

Engineering Software Product Lines may be a strategy to reduce costs and efforts for developing software and increasing business productivity. However, it cannot be considered as a "silver bullet" that applies to all types of organizations. Companies must consider pros and cons to determine sound reasons and justify its adoption. In previous work, we proposed the APPLIES evaluation framework to help decision-makers find arguments that may justify (or not) adopting a product line engineering approach. This paper presents our experience using this framework in a mid-sized software development company with more than 25 years of experience but without previous experience in product line engineering. This industrial experience, conducted as a qualitative empirical evaluation, helped us to evaluate to what extent APPLIES is practical to be used in a real environment and to gather ideas from real potential users to improve the framework.

CCS CONCEPTS

• **General and reference** → **Cross-computing tools and techniques**; **Empirical studies**; • **Software and its engineering** → *Software product lines*;

KEYWORDS

Empirical evaluation, product line adoption, product line engineering, qualitative evaluation

ACM Reference Format:
Luisa Rincón, Raúl Mazo, and Camille Salinesi. 2019. Analyzing the Convenience of Adopting a Product Line Engineering Approach: An Industrial Qualitative Evaluation. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3307630.3342418

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342418

1 INTRODUCTION

A product line is a set of similar products that share common characteristics, meet the requirements of a market segment and are implemented from a common set of core assets in a prescribed way. Product Line Engineering (PLE) is a systematic and comprehensive approach for developing and maintaining product lines [5].

Adopting a product line engineering approach may seem very attractive to software companies due to the benefits this paradigm promotes such as reduced costs, reduced time to market, or improved quality [5, 25]. However, despite those promised and proved benefits, researchers have identified that adopting a product line engineering approach involves barriers that not all companies are prepared to face. Some of these barriers are, for example, the time required in the adoption process, the costs involved in establishing the product line, the new practices, processes or training required and the cultural resistance that is normal to encounter when organizational changes are introduced [4, 8, 12]. Not all circumstances justify the adoption of the PLE approach and therefore, to better overcome the barriers, decision-makers must justify this change with well-founded arguments.

One way to find these well-founded arguments is to assess the convenience of adopting the PLE approach in the enterprise before going any further in the adoption initiative. In fact, in the literature different authors encourage to evaluate the readiness of the company before introducing any organizational change [1, 10, 13]. In the product line literature, contributions often focus on technical issues (e.g. variability modeling, variability reasoning, product line architecture, product configuration or product derivation), while less attention has been paid to organizational and process-related issues such as product line adoption [19]. Exceptions include approaches such [9, 18, 26].

To reduce this gap, in previous work we proposed APPLIES, a framework for *evaluAting organization's motivation and Preparation for adoPting product LInES*[21, 22]. This framework makes it possible to assess, on the one hand, whether the company is motivated to adopt a PLE and, on the other hand, to what extent the company has operational, technical and economic factors that denote its readiness to adopt a PLE approach.

We have carried out a series of evaluations following a design-science methodology [29] to evaluate and improve APPLIES based on the results of different evaluation scenarios. So far, it has been

evaluated by five academic experts and an industrial with experience in product line engineering [20]. In addition, APPLIES was evaluated by 14 potential users in the frame of a quasi-experiment [22].

This paper reports the second experience in evaluating APPLIES in an industrial environment. In this experience we evaluated APPLIES (version 2.0.a1) in a software company with more than 25 years of experience in the development of custom software solutions, but no experience in product line engineering. This empirical evaluation had the purpose of evaluating the usefulness that potential users found in APPLIES, the understandability of its content, and the suggestions for improvement that potential users suggested after using our proposal. Compared to the first evaluation, both evaluations had the same research questions but the stakeholders that participated in both experiences as well as the characteristics of the company were different. In the first industrial evaluation the company had already created a product line. The CEO used APPLIES to evaluate retrospectively the convenience of adopting product line engineering [20]. Conversely, in the evaluation reported in this article, the company has no previous experience in product line engineering and although it was interested in increasing the reuse it had not previously considered the PLE. In addition, only the CEO participated in the first industrial evaluation, while four company leaders participated in the evaluation here reported.

Also, this article presents the process we used to analyze the qualitative data. We expect that these methodological details may be useful to other researchers interested in conducting qualitative research in product lines.

The remainder of this paper is organized as follow: Section 2 introduces the APPLIES Framework. Section 3 describes the experimental design of the empirical evaluation and discusses the process applied to analyze the results. Section 4 summarizes the findings. Section 6 presents the threats to validity. Section 7 discusses related work. Finally, Section 8 contains some concluding remarks, shows the lessons learned during the process and presents our directions to future work.

2 APPLIES FRAMEWORK

APPLIES is an evaluation framework that assists decision-makers to examine the convenience of adopting a product line engineering approach in an organization.

Mainly, APPLIES helps stakeholders to identify and prioritize the drivers that motivate the adoption of software product lines, assess to what extent a company is prepared to adopt a product line approach, and identify those factors that require more attention.

The results of the analysis are presented in the form of charts summarizing the information. Decision-makers can use this information to identify the reasons why product line engineering would or would not be convenient for their business. Space restrictions prevent the full presentation of APPLIES in this article, but full details are available in previous publications [21, 22].

How to use it?

The part of APPLIES visible to users is an Excel workbook in charge of capturing user responses, automatically processing them, and plotting the results. This tool is available for download on the project website[1]. Decision-makers can use this workbook to: (i) analyze a hierarchical set of factors and (ii) provide yes/no answers (§Fig.1a) and values in a Likert-response format (§Fig.1b) for a well-defined set of statements (78 in total) organized in dimensions of analysis. After gathering data from users, the Excel book automatically summarizes the results and presents quantitative data and graphs regarding the organization's performance in the analyzed dimensions (§Fig.1c and 1d).

Audience

Consultants, project managers or anyone who needs information to convince management levels to adopt a product line initiative.

Previous knowledge

APPLIES asks basic product-lines related concepts, such as "potential products", "domain", "reusable artifacts". Users should therefore have at least a basic notion about basic product line engineering concepts, e.g. benefits, drawbacks, characteristics, etc.

3 EVALUATION DESIGN

3.1 Purpose and research questions

With an exploratory purpose, the main objective of this empirical study was to evaluate the APPLIES framework in the context of a real-world software company that has no experience in the software product line field. The idea was to collect stakeholders' perceptions regarding the proposed framework and their expectation for improving it.

Based on this objective, the following research questions (RQ) guides this empirical evaluation:

- RQ1: How useful do practitioners perceive the APPLIES framework and why?
- RQ2: How understandable is the content of the APPLIES framework for the users? Why?
- RQ3: What additions or modifications could improve the APPLIES framework?

We will answer these research questions from qualitative information collected from participants' opinions and observations made by the researcher during the empirical experience

3.2 Company background

Asesoftware is a multinational company with clients in Colombia, Costa Rica, United States and Chile. This company has more than 25 years in the software services industry and more than 250 employees. Among its services, Asesoftware offers custom software solutions, specialized consulting, data analysis solutions and IT consulting. Since 2010 Asesoftware has been certified with level 5 for the CMMi-DEV model and it obtained level 3 for the CMMi-SVC model in 2018 [2]. Asesoftware was selected for this evaluation based on its suitability. On the one hand is a software development company, which is the target audience of APPLIES. On the other hand, the company agreed to use APPLIES to assess its motivation and preparedness to adopt a product line engineering approach.

Why is a product line initiative interesting? According to the company's CEO, currently, Asesoftware reuses less than 10% of the artifacts already developed but would like to increase its reuse

[1] www.applies.variamos.com
[2] https://asesoftware.com

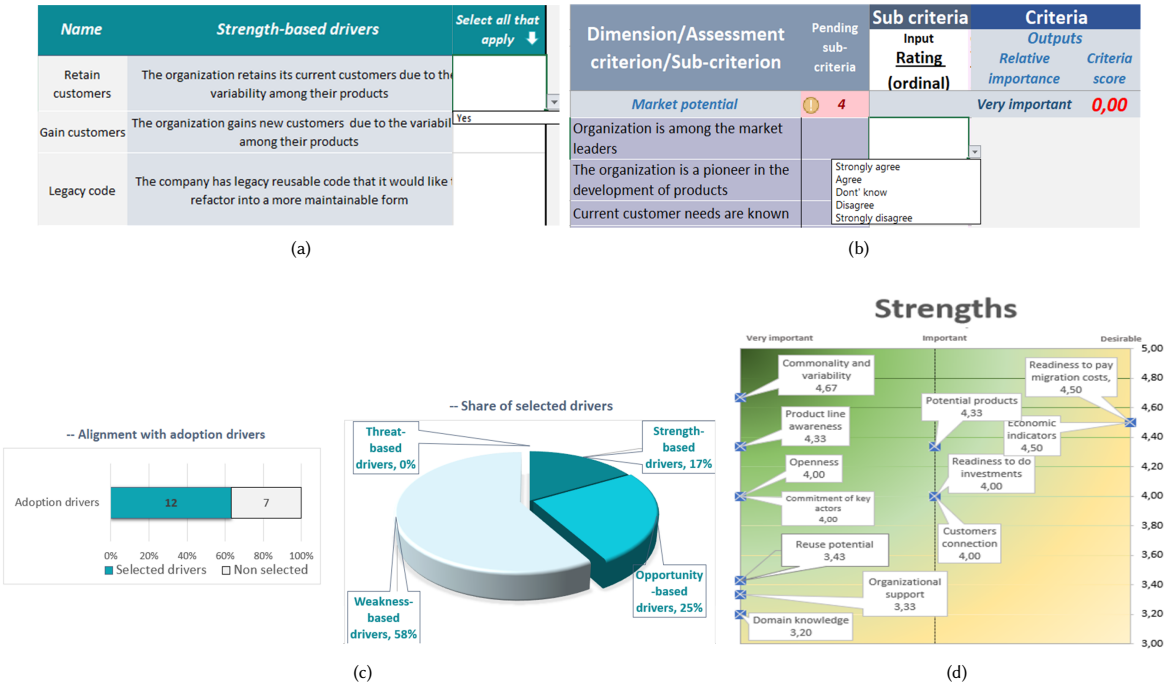


Figure 1: Screen-shots. Spreadsheet-based tool that supports APPLIES

rate to more than 50%. The CEO believes this is possible because the company often develops software components that can be shared across different domains for different applications, *e.g.*, modules for managing addresses, dates and holidays.

3.3 Design

The evaluation was conducted in two workshop sessions of two hours each. Both workshops took place at the company's facilities in May 2018. Four executives who have been working in the company for more than eight years attended both sessions: the leader in architecture, the leader in innovation, the leader in knowledge management and the CEO. All these participants had sufficient knowledge of the company's current and expected conditions due to their years of experience in the company and their managerial position. Therefore, all of them were qualified for using APPLIES in Asesoftware.

Both sessions were moderated by one of the authors of this paper. The researcher, who has been not involved in company projects, lead the sessions, moderated the discussion and, when necessary, answered questions, or clarified the content of the framework.

We used observation and focus group to collect data [23]. The observation took place while each participant individually completed the tasks assigned by the researcher, whereas the focus group took place at the end of each session. We prepared as handouts: (i) slides introducing the product line engineering approach and presenting the APPLIES framework; (ii) a questionnaire to get feedback from the participants after using APPLIES; and (iii) the tasks to be performed by the participants. These tasks were to use APPLIES to assess the motivation and preparation that Asesoftware had to adopt a product line engineering approach [3].

3.4 Conduction

The *first workshop session* had three parts: context, execution and discussion.

Context: the researcher introduced himself and asked the four company participants to introduce themselves. The researcher then gave a brief presentation about product line engineering to refresh the participants' memories, as they were all familiar with the topic. Afterward, the researcher presented the purpose of the evaluation to be carried out, presented the generalities of APPLIES and explained in detail the part of APPLIES that helps the company to evaluate its motivation to adopt a product line engineering approach.

Execution: the researcher asked the participants to use the Excel workbook that implements APPLIES to assess Asesoftware's motivation to adopt a product line engineering approach. The tool had been previously emailed to each participant at the start of the session. Each participant performed the ratings independently. Sometimes the participants asked the researcher questions about the content of the framework. In those cases, the researcher resolved the doubts and took notes to take into account which of the factors included in the framework needed improvements to clarify its content.

[3]The material related to this empirical experience is available at https://applies.variamos.com/evidence/V20a1

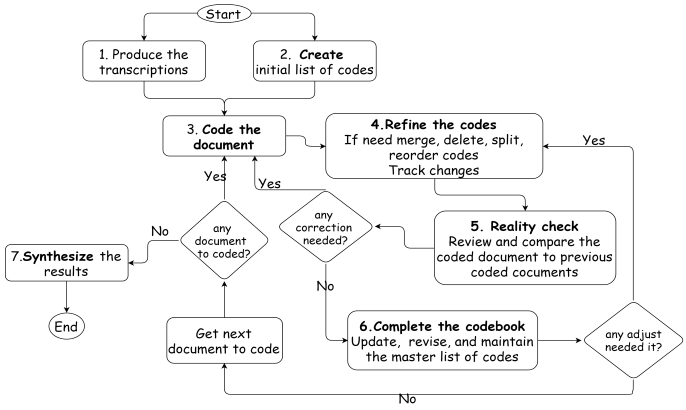


Figure 2: Data analysis and coding process

Discussion: the researcher started the focus group when all participants finished completing the questions regarding the motivation. This focus group was recorded with the consent of the participants, and the recordings were analyzed to answer the research questions (see Section 3.5). The focus group had the following parts: initially, each participant indicated the answer he gave to each question. When there were differences in the participants' responses, the researcher moderated a discussion in which the participants justified the reasons for their selection until consensus was reached. Furthermore, in cases where the answers were different due to divergences in the interpretation of the content of the question, the researcher clarified the meaning and moderated the discussion, if appropriate, in order to obtain a consensus on the answer. These cases were taken into account to assess the understandability of the framework content as described in Section 4. Before closing the session, the researcher led a discussion for (i) reviewing the results provided by APPLIES, (ii) capturing participant's opinion about their experience using APPLIES, and (iii) capturing participant's recommendations for improving this framework.

The *second workshop session* also had three parts. During the ***Context*** the researcher presented APPLIES-preparation and during the ***Execution and discussion*** the researcher conducted the same activities as in the first session, but the participants interacted with the part of APPLIES that evaluates company's preparation. Therefore, the questions and comments received were related to this part of the framework.

3.5 Analysis procedure

Figure 2 presents the process followed to analyze the collected data. This process was inspired by the guidelines available in [14, 24] and was supported by MAXQDA [4], a Computer-Assisted Qualitative Data Analysis Software that assists researchers for transcribing, coding, and later analyzing qualitative data. The first author conducted this analysis under the direction of the second and third authors. Section 6 presents the strategies used to mitigate the biases resulting from this individual review.

The remainder of this section presents the details of the data analysis procedure.

[4]https://www.maxqda.com/

Step 1. Produce the transcriptions: Transcribing means converting an audio or video file to a written format [2]. Recordings were transcribed into a written form because in this format we can study them in detail. Both sessions of the workshop were conducted in Spanish, the transcripts were therefore prepared in the same language. The audio files were transcribed following an "intelligent transcription" strategy *i.e.,* a transcription that intends to improve legibility of the text by *e.g.,* by solving grammatical errors, removing irrelevant words or sentences and adding personal pronouns or articles omitted in the rushed speech. Intelligent transcriptions are useful for academic purposes where legibility and clarity are fundamental [2].

Step 2. Create the initial list of codes: a code is a "tag or label for assigning units of meaning to the descriptive or inferential information compiled during a study" [24]. The coding process started with a list of codes compiled from evaluations previously reported in [20, 22]. The coding process allows researchers to think from raw data up to conceptualization and connection between ideas and concepts. For this reason, coding is an iterative process by nature [24] where the initial list of codes was updated iteratively. For example, Figure 3 presents a fragment of the initial and final list (§see Fig.3a and §Fig.3b respectively). As can be seen in the figure, four codes were added between the initial version and the final version.

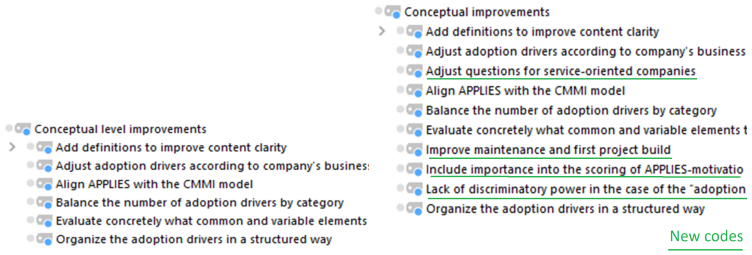
Step 3 and 4. Code the document and refine the codes: Codes were assigned to fragments of raw data such as phrases, sentences, or paragraphs. Also, as the coding progressed, we created more abstract codes, i.e., subcategories and categories, to organize concepts thematically. Figure 3c shows an example of an encoded transcript. The text on the left side of the margin is a code assigned to a transcript extract.

Step 5. Reality check: Each time a transcript was completely coded, it was read again to verify the consistency between the fragments and the assigned codes. Additionally, when code modifications affected fragments that had already been coded, these were rechecked to ensure that the assigned codes were still consistent with the text. In this process, the MAXQDA tool was handy because it offers functionalities to merge, eliminate, divide, or reorganize the code hierarchy.

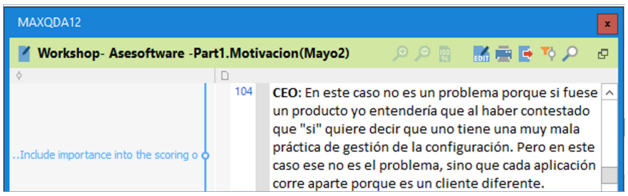
Step 6. Complete the codebook: The codebook is a document that helps those who codify the material to maintain consistency in the coding process [24]. This document contains the list of codes used during the coding process and the explanation of the meaning of each code and is built iteratively along with the coding process whereby its information is completed and refined as the material is reviewed [24]. There is a codebook that contains the codes that have arisen from the analysis of the material collected in the different empirical evaluations of APPLIES.

Step 7. Synthesize the results: In this step the coded transcripts were analyzed to summarize the results. We used the "summary grid" functionality [5] provided by MAXQDA. This functionality facilitates to visualize in one place segments tagged with each code and offers the possibility to sum up abstractly what the transcripts say.

[5]More info available at https://www.maxqda.com/help-max18/summary-grid/creating-and-editing-summaries



(a) Extract - initial list of codes (b) Extract - final list of codes



(c) Example of coding in MAXQDA

Figure 3: Coding examples. Screen-shots from MAXQDA

4 RESULTS

This section presents the results of the three research questions presented in Section 3. Excerpts of the transcripts were translated from Spanish into English to better illustrate the results.

4.1 RQ1: Perceived usefulness

Our results evidenced that APPLIES is perceived as a useful tool applicable in real scenarios. The participants considered that APPLIES would "*allow them to assess whether or not the company would be motivated to adopt the product lines.*" However, some adoption drivers may require adjustments as not all would be sufficient indicators to claim that product lines are the best solution for the company, as is explained in Subsection 4.3. Regarding the preparation part, the participants considered that the information provided by APPLIES would guide them to know which aspects they should analyze most carefully before proceeding with the decision of adopting or not such development approach. Here are some comments to highlight users' perceptions about the outputs of the framework: "*I liked that a summary appears at the end (referring to the chart presented in Figure 1d). I want to see in black and white if the product lines would be adequate and the numbers on the chart are clear to me.*" Another participant said, "*It seems to me that the map and the questions place us in the weak points that need to be attacked and in the strength that we would have. It gives us a panorama that seems clear to me of the current situation to initiate a product line.*"

4.2 RQ2: Content understandability

The participants found APPLIES easy to use and follow. For example, one of them said: "*Although there were some questions that were not entirely understandable, I would say that APPLIES was generally easy to use.*" In terms of content clarity, participants struggled to interpret seven questions included in APPLIES. To analyze how we can make the questions clearer, we recorded participants' feedback about these questions. Two questions concerning the economic dimension were difficult to assess, as the company has not yet adopted the product line engineering approach. We plan to adjust these statements because APPLIES would be used by companies that have not decided to proceed with the adoption of the PLE and therefore this same difficulty would be experienced by other companies while using the framework. Furthermore, one participant stated that APPLIES had items asking in the same question whether the company knew about the product lines or whether it would be willing to learn about them, which evaluates two different things. We found only one question that had this problem and should be reviewed.

A final remark regarding the content understandability is that Asesoftware builds and maintains tailor-made software, i.e, the company does not have a main software product that commercializes to different clients, nor a specific market domain in which it limits its services. In this regard the participants suggested to adjust the questions depending on whether the company is focused on providing IT services, or is focused on creating and customizing products for a particular market segment.

4.3 RQ3: Improvements

Response format: We noticed participants understood the valuation methodology for APPLIES-motivation and APPLIES-preparation. As mentioned in Section 2 users should provide yes/no answers (§Fig.1a) and values in a Likert-response format (§Fig.1b). However, one participant suggested unifying the methodology of evaluation. He said: "*It seems to me that there is a high contrast between the two evaluation methodologies... I think it is better to have only one.*" According to the comments received from the participants, a response format that includes more choices, such as a Likert scale would reduce response bias. The participants expressed they felt induced to answer "yes" in some questions of APPLIES-motivation.

We observed this was especially true in the case of adoption drivers related to the company's strengths and opportunities. Also, we detected the need of including an additional option into the Likert scale for representing an intermediate value between "agree" and "disagree." Particularly, there were three sub-criteria in which participants felt the need for this intermediate value, but it did not exist.

Include a mechanism for reaching consensus: This experience was the first case in which different people from the same company used APPLIES simultaneously. This way of using the framework pointed out to us that it is important to include guidelines that lead people to reach a consensus on the answers. For example, although all participants in the evaluation answered the same questions, the responses varied on average in 50% of the cases. It was during the review of the results that a consensus was reached to define a single answer for each question.

One participant also said that it would be important to include guidelines in the framework indicating when it is appropriate to select each option from the scale of responses. For example, during the evaluation we observed that participants used "strongly agree" when in their opinion the company fully complied with the statement of the question, while they selected "agree" when they felt that the company complied with what was stated in the question but could still improve. A guide that generalizes the behavior observed in Asesoftware could guide APPLIES users to choose their answers with a common frame of reference, which in turn could help to facilitate consensus in the answers.

Proposed new content. Two new assessment criteria appeared during the discussions: *evaluate the impact that reuse practices would have inside the organization* and *evaluate if the company has the freedom from legal constraints that might restrict the reuse of artifacts between projects or products*. Also, one participant expressed that it would be useful that APPLIES provides additional guidance regarding the adoption. He said: *"there is still some uncertainty about what is the next step to do if we decide to elaborate a product line engineering approach"*.

Tool. The four participants completed all assigned tasks and had no critical errors using the spreadsheet tool that implements APPLIES. However, one of the participants expressed that he would prefer an Excel sheet that shows only the data needed to conduct the evaluation.

Review the pertinence of the adoption drivers. The CEO said, *"I am left with the doubt of when does one say no? ...Our business is custom software development, so everything that has to do with software development fits, and everything that has to do with new technologies in software development also fits our case*." The adoption drivers included in APPLIES-motivation should assist companies to differentiate when they are motivated to adopt a product line engineering approach. However, the comments collected from the participants lead us to think that companies for which product lines are not necessarily interesting could answer "yes" to some drivers included in APPLIES-motivation. Some of the adoption drivers in which we detected this problem are: "*Technological advances make possible to migrate existing products with heterogeneous technology to the same technology*," "*There are overlapping elements in the plans for different products, e.g., upcoming trends or domain-specific technologies that are expected to be used within many products in the future*." We plan to review the adoption drivers considered in APPLIES-motivation because the purpose of this part of the framework is to identify to what extent the needs of the company are aligned with the objectives that a product line engineering could tackle.

Need to consider both the importance and score of the adoption drivers. We summarized this finding as the need to include in the assessment of the company's motivation a way to evaluate the importance of each adoption driver. In this way, it would be possible to differentiate to what extent a company wants to pay attention to the factors that might drive the adoption. For example, APPLIES-motivation has the following adoption driver: "*Product variants are implemented in source code files that are scattered in different parts of the code repository*." If people select "yes", it means that there is a motivation for adopting product line engineering. However, one of the participants said: "*The innovation leader selected "Yes" because nothing is really controlled here, each project goes its own way, but in our case it doesn't matter because each case is a different product. So I selected "no", but not because we control the variants, but because in our case everything runs separately. Each client has its own source and this is not a problem* ."

5 LESSONS LEARNED

Use the right tools for the job: based on our experience, we strongly recommend the use of computer-assisted qualitative data analysis software (CAQDAS) to transcribe, code, and analyze the material. These types of tools help to maintain the organization of the research, stay linked to the original text segment from the raw data, and record the analysis of the data. Also in our case MAXQDA was very useful to produce the transcripts. When preparing the transcripts, it is necessary to stop, advance and return the audio as the text is produced. Without the right tool, activities such as manually stopping and rewinding recordings are laborious because rewinding the recording can leave the audio behind or ahead of what is needed. This lack of support wastes the time of those who produce the transcripts. Specialized tools for these tasks offer features to easily transcribe audios, such as playback speed, rewind interval, and time-stamps to synchronize transcripts with audio or video files. There are paid and free programs for qualitative analysis, some of them are for example NVivo, ATLAS.ti, Dedoose and MAXQDA.

Researchers need to decide how they will represent audible data in a written form: transcribing seems to be a simple technical task. However, it involves decisions about how the audible data will be represented in the text, e.g., inaudible material, quotations, non-verbal communication, pauses, elements to omit, etc. We advise researchers to identify what strategy they will use to produce the transcripts and to develop guidelines detailing the rules to be followed in that process. As a guide, we consult the following sources of information on how to prepare transcripts [2, 6, 11, 16, 27].

Transcribing takes a long time: although the preparation of transcripts could be left to external research staff, we decided to create them ourselves because "*this is an important first step in data analysis[...] it can facilitate the realizations or ideas that arise during analysis*" [2]. One limitation to consider, however, is that producing transcripts is a time-consuming task. According to the literature, transcribing an hour of talk at a general level of detail

takes at least three hours and can go up to ten hours per hour with a fine level of detail [2]. It took us 17 hours to transcribe three hours and 20 minutes of recording, approximately six hours per hour. We thought we should better plan what information is worthy of transcribing to reduce this time. For example, we would not transcribe participants' explanations of the answers given to APPLIES questions or the discussions that took place among participants to reach consensus on the answers. Instead, we would only transcribe those parts of the audio that were relevant for answering the research questions, such as the difficulties expressed by the participants in understanding the content of APPLIES, the opinions they expressed about the usefulness of the framework or about the changes that could improve it.

6 THREATS TO VALIDITY

There are some attention points that could potentially limit the scope of the results of this research.

Internal validity: it is affected when a researcher has not adequately controlled interfering variables or was not aware of them at all. This lack of control can influence the outcomes of the evaluation, which can lead to false cause/effect relationships. To ensure the rigor of the evaluation process, we defined and followed a protocol which is based on well-known guides of empirical research [7].

External validity: is concerned with the generalization of the results. Four members of a software company not randomly chosen were studied, then we cannot make statistical generalizations about the quality attributes measured in the industry in general. However, these findings help us to find points of improvement of APPLIES and complement evaluation results obtained from other evaluations settings [20, 22].

Construct validity: this threat refers to ensure the correctness of the measures involved in the investigation, the relevance of the used concepts and the proper chain of evidence. Construct validity was mitigated with the following strategies:

- *Effects of one interpretation of one single data source:* the evaluation took place in a single company but four different people participated in it, so we considered that the interpretation bias from a single source was reduced. Additionally, extracts from the transcriptions that justify the reasoning behind the resulting analysis were kept. In this way we keep the link between the data source and the corresponding interpretation.
- *Amount of false or incomplete information:* we cross-checked recordings and notes from observations for contradictions to reduce the amount of false or incomplete information.
- *Avoid misconceptions and misunderstandings between the collected data and the reality:* we used the strategy proposed by Runenson et al. [23] in which during the workshop sessions the researcher paraphrased and summarized the answers given by the participants in order to confirm that the opinions had been correctly understood.
- *Avoid inconsistency in the analysis process:* we designed a set of guidelines to process the collected transcripts[6]. Also as recommended in [24], we created a codebook with definitions and examples that help us to maintain consistency in the coding process.
- *Test-retest:* this strategy involves repeating (after an appropriate time-frame) some or all the actions that were taken into the study to compare the outcomes. This strategy was used during the transcript and coding process to control subjectivity and bias. For example, a week passed between the completion of the first version of the transcripts and its second revision and adjustment. Regarding the coding of the material, a week also passed between the initial coding of the transcripts and the revision of the coding before proceeding with the analysis and interpretation of the results. Also, the second author randomly reviewed coded data segments to assess the relevance of the assigned codes.

[6] Available online at http://doi.org/10.13140/RG.2.2.30403.66086/1

7 RELATED WORK

Mazo et al. [15] present some preliminary processes to be carried out before adopting (or not) a product line production strategy. Nazar et al. [17] present an ethnography in which they analyze the potential of a Chinese company to adopt product line engineering practices. The authors analyze the data in a qualitative manner and based on their results suggested improvements. In this article we also address company's potential for adopting product line engineering, but we present a qualitative evaluation of a framework that assists companies in evaluating the convenience of adopting product line engineering.

In "the early years" of product line research, some approaches were proposed around organizational and process-related topics such as product line adoption. For example, Bandinelli and Sagardui [3] propose an overview of the benefits and risks that a product line adoption might imply. Schmid and John [26] define a method to evaluate whether a domain has sufficient potential for reuse based on structured interviews. Fritsch and Hahn [9] propose a method that analyzes the target market and the potential products of a company to evaluate whether a systematic product line development would be helpful for an organization or not. The software engineering institute proposes the "Product Line Technical Probe" [18]. This method performs a diagnostic based on the 29 practice areas specified in the SEI Framework to examine the preparation that a company has to succeed with a software product line approach. Finally, more recently, Tüzün et al. [28] propose a decision support system to help companies to select what transition strategy will help them to migrate towards a product line approach.

APPLIES is complementary to these existing approaches. On the one hand, APPLIES-preparation includes the criteria proposed by the previous approaches but also considers other factors retrieved from the literature. Furthermore, APPLIES-preparation provides a defined process and a tool to operationalize the assessment. On the other hand, unlike the previous approaches, APPLIES-motivation also analyzes the motivation of a company for adopting a product line approach.

8 CONCLUSIONS

This article presents the evaluation of APPLIES (version 2.0.a1) in an industrial context in which four leaders from Asesoftware, a

company providing IT services and customized software solutions used the APPLIES evaluation framework to evaluate the convenience of adopting product line engineering in their organization. This company focuses its efforts on creating and maintaining technological solutions that meet the needs of its customers. In this process the company builds software solutions but does not have a software product that commercializes to different clients, nor a specific market domain in which it limits its services. This orientation towards offering services and not products helped us find issues in APPLIES that we had not identified in previous evaluations.

The empirical evaluation consisted of two workshop sessions in which we collected data through observation and focus groups. This paper details the procedure for preparing and analyzing the data collected, as well as lessons learned about the process of transcribing data from audio recordings. We expect that these procedural details will be useful to other researchers interested in conducting qualitative research in product lines engineering.

The results of this experience show us that the participants considered that APPLIES is useful to identify the reasons why it would be interesting for Asesoftware to adopt a product line engineering approach, as well as the strengths and weaknesses that the organization could face if it decides to adopt this approach. In terms of comprehensibility, we identified questions that we must adjust to avoid ambiguity, others that we must divide and others that we must reconsider, as users would not have the information to answer them. With the data collected we identified, among others the following ways to improve the proposal: to adjust the scale that evaluates the motivational part and to incorporate an intermediate value between "agree" and "disagree" in the scale that evaluates the preparedness. Furthermore, we identified the need of excluding those questions that are not exclusive to motivate a product line engineering approach and incorporating a mechanism to facilitate consensus between answers, when multiple people use APPLIES to evaluate the same case.

We are currently comparing and contrasting the results obtained in the collection of empirical evaluations carried out so far to decide which adjustments we will incorporate into the next version of the framework.

ACKNOWLEDGMENTS

We would like to extend our gratitude to the participants from the company who dedicated their time and effort to participate in the empirical evaluation reported in this paper.

REFERENCES

[1] Achilles A. Armenakis, Stanley G. Harris, and Kevin W. Mossholder. 1993. Creating Readiness for Organizational Change. *Human Relations* 46, 6 (1993), 681–703. https://doi.org/10.1177/001872679304600601 arXiv:0803973233

[2] Julia Bailey. 2008. First steps in qualitative data analysis: transcribing. *Family Practice – an international journal* 25, 2 (feb 2008), 127–131. https://doi.org/10.1093/fampra/cmn003

[3] Sergio Bandinelli and Goiuri Sagardui Mendieta. 2000. Domain Potential Analysis: Calling the Attention on Business Issues of Product-Lines. In *Lecture Notes in Computer Science*. Vol. 1951. 76–81. https://doi.org/10.1007/978-3-540-44542-5_9

[4] Cagatay Catal. 2009. Barriers to the adoption of software product line engineering. *ACM SIGSOFT Software Engineering Notes* 34, 6 (2009), 1. https://doi.org/10.1145/1640162.1640164

[5] Paul Clements and Linda M Northrop. 2001. *Software Product Lines: Practices and Patterns* (1st ed.). Addison-Wesley Professional.

[6] Christina Davidson. 2009. Transcription: Imperatives for Qualitative Research. *International Journal of Qualitative Methods* 8, 2 (2009), 35–52. http://journals.sagepub.com/doi/10.1177/160940690900800206

[7] S Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering*. 285–311. https://doi.org/10.1007/978-1-84800-044-5_11

[8] 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 *Proceedings of the15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011)*, Vol. 2011. IET, 11–20. https://doi.org/10.1049/ic.2011.0002

[9] Claudia Fritsch and Ralf Hahn. 2004. Product Line Potential Analysis. In *International Systems and Software Product Line Conference (SPLC)*. 228–237. https://doi.org/10.1007/978-3-540-28630-1_14

[10] Daniel T. Holt, Achilles A. Armenakis, Hubert S. Feild, and Stanley G. Harris. 2007. Readiness for organizational change: The systematic development of a scale. *Journal of Applied Behavioral Science* 43, 2 (2007), 232–255. https://doi.org/10.1177/0021886306295295

[11] Áine Humble. 2016. Guide to Transcribing. *Department of Family Studies and Gerontology, Mount Saint Vincent University* (2016), 1–3. http://www.msvu.ca/site/media/msvu/TranscriptionGuide.pdf

[12] M.A Jabar, M.B. Zarei, F Sidi, and N.F.M Sani. 2013. A review of software product line adoption. *Journal of Theoretical and Applied Information Technology* 57, 1 (2013), 88–94.

[13] John P. Kotter. 2002. *The Heart of Change*. Harvard Business School Press, Boston, Massachussets.

[14] Margaret D. LeCompte. 2003. Analyzing qualitative data. *Theory into Practice* 39, 3 (2003), 12.

[15] Raúl Mazo, Gloria Giraldo, and Germán Urrego. 2018. Estudio de mercado y de factibilidad para proyectos de líneas de productos. In *Guía para la adopción industrial de líneas de productos de software*, Raúl Mazo (Ed.). Editorial Eafit, Medellín-Colombia, 61–80.

[16] Minnesota Historical Society Oral History Office. 2001. *Transcribing, Editing and Processing Guidelines*. Technical Report. Oral History Office, Minnesota Historical Society. www.mnhs.org/.../ohtranscribing.pdf

[17] N. Nazar and T. M. J. Rakotomahefa. 2016. Analysis of a Small Company for Software Product Line Adoption — An Industrial Case Study. *International Journal of Computer Theory and Engineering* 8, 4 (aug 2016), 313–322. https://doi.org/10.7763/IJCTE.2016.V8.1064

[18] Linda Northrop, Larry Jones, and Patrick Donohoe. 2005. *Examining product line readiness: experiences with the SEI product line technical probe*. Technical Report. Sofware Engineering Institute.

[19] 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 *Proceeedings of the 22nd International Conference on Systems and Software Product Line - SPLC '18*. ACM Press, New York, New York, USA, 14–24. https://doi.org/10.1145/3233027.3233028

[20] Luisa Rincon, Jaime Chavarriaga, Raul Mazo, and Camille Salinesi. 2018. How Useful and Understandable is the APPLIES Framework? a Preliminary Evaluation With Software Practitioners. In *Proceedings of the ICAI Workshops (ICAIW)*. IEEE, 1–6. https://doi.org/10.1109/ICAIW.2018.8555002

[21] Luisa Rincon, Raul Mazo, and Camille Salinesi. 2018. APPLIES: A framework for evaluAting organization's motivation and preparation for adopting product lines. In *Proceedings of the 12th International Conference on Research Challenges in Information Science (RCIS)*. IEEE, 1–12. https://doi.org/10.1109/RCIS.2018.8406641

[22] Luisa Rincon, Raul Mazo, and Camille Salinesi. 2018. Evaluating Company ' s Readiness for Adopting Product Line Engineering : a Second Evaluation Round. *Complex Systems Informatics and Modeling Quarterly* 17 (2018), 69–94.

[23] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. 2012. *Case Study Research in Software Engineering: Guidelines and Examples* (1st ed.). Wiley Publishing.

[24] Johnny Saldaña. 2013. *The Coding Manual for Qualitative Researchers*. 329 pages. https://doi.org/10.1109/TEST.2002.1041893 arXiv:arXiv:gr-qc/9809069v1

[25] Klaus Schmid and Isabel John. 2001. Product line development as a rational, strategic decision. *Proceedings of the International Workshop on Product Line Engineering in Early Steps: Planning, Modeling, and Managing (PLEES'01)* (2001), 1–6.

[26] Klaus Schmid and Isabel John. 2002. Developing, validating and evolving an approach to product line benefit and risk assessment. In *Proceedings of the 28th Euromicro Conference*. 272–283. https://doi.org/10.1109/EURMIC.2002.1046172

[27] Transcribe.com. 2019. Style guide. (2019).

[28] Eray Tüzün, Bedir Tekinerdogan, Mert Emin Kalender, and Semih Bilgen. 2015. Empirical evaluation of a decision support model for adopting software product line engineering. *Information and Software Technology* 60 (apr 2015), 77–101. http://dx.doi.org/10.1016/j.infsof.2014.12.007

[29] Roel J. Wieringa. 2014. *Design Science Methodology for Information Systems and Software Engineering*. Vol. 2. Springer Berlin Heidelberg, Berlin, Heidelberg. 493 pages.

Identifying Collaborative Aspects During Software Product Lines Scoping

Marta Cecilia Camacho Ojeda
cecamacho@unimayor.edu.co
Institución Universitaria Colegio Mayor del Cauca
Popayán, Colombia

Francisco Álvarez Rodriguez
fjalvar@correo.uaa.mx
Universidad Autónoma de Aguascalientes
Aguascalientes, México

César A. Collazos
ccollazo@unicauca.edu.co
Universidad del Cauca
Popayán, Colombia

ABSTRACT

The software product line engineering (SPLE) is a reuse strategy that allows software companies to save effort when they develop products with common features. There, the software product line scoping is one of most essential and complex activities because (1) a correct scope for the line has a high impact in its success and (2) it implies an interdisciplinary activity involving stakeholders with different visions about the products. In this paper, we report an exploratory study aimed to identify problems related to the collaborative work at scoping SPL in practice. We studied problems related to the participation and interaction of stakeholders in projects where groups of students must develop SPLs of serious video games for training employees in a company. Our study revealed problems related to low levels of communication, participants with different project objectives and stakeholders requesting different types of programs. Problems that are exacerbated by the staff rotation and inconveniences scheduling working sessions. In addition, our study revealed other problems regarding developers misunderstandings the artifacts related to the scope and their use in the further development activities. In this paper, we also present the first version of a collaborative method for SPL scoping, which seeks to combine scoping practices with collaborative patterns and thinkLets, with this combination we seek the effective participation of the required roles in this activity.

CCS CONCEPTS

• **Software and its engineering** → *Software product lines*; *Collaboration in software development*;

KEYWORDS

Software Product Lines, SPL Scoping, Collaborative Work, Collaboration Engineering

ACM Reference format:
Marta Cecilia Camacho Ojeda, Francisco Álvarez Rodriguez, and César A. Collazos. 2019. Identifying Collaborative Aspects During Software Product Lines Scoping. In *Proceedings of 23rd International Systems and Software Product Line Conference, Paris, France, 9–13 September, 2019 (SPLC'19),* 8 pages.

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

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.1145/3307630.3342420

https://doi.org/10.1145/3307630.3342420

1 INTRODUCTION

A software company may apply *Software Product Line Engineering (SPLE)* as a strategy to increase its productivity and competitiveness. A *Software Product Line (SPL)* is a set of products that share common features, where each product satisfies a specific market and is developed from a common collection of assets [9].

SPL scoping is one of the most relevant activities of SPLE [10]. scoping defines which products are part of the line and which ones are not, delimit the functional domains and raises the components should be developed in a reusable manner [27]. Only that scoping is not an easy task, it requires the participation of people with different knowledge [17][27]. If the scoping is performed only by technical people, the defined products may be not suitable for the market. If it involves only experts in the domain, it is possible that production effort can be very high and if a scope defined without customer representatives, products may not useful by end users. Thus SPL scoping is an interdisciplinary activity [1] [6] [9].

The scoping needs to be understand as a collaborative activity because it is very difficult that a single participant has all the necessary knowledge to perform the activity in isolation [22]. Regretfully, the interdisciplinary work required for scoping is hard to achieve because each participant may come from different areas, each one with its own language, concerns and interests about the product line [26]. It is important to overcome diverse communication and collaboration issues to achieve an effective participation of all the stakeholders, where a common vision of the SPL project is obtained through exchanging knowledge and negotiating proposals of the participants [22][26].

This paper reports an exploratory study aiming to identify which scoping tasks require collaborative practices and how the process can be improved by integrating these practices. This study, where we identified problems analyzing diverse software product lines proposed by groups of students define the scope for a company, has been used as a foundation to improve the scoping tasks in *Small SPL*, an engineering process for product line engineering. The resulting proposal integrates a selection of Thinklets[4][13], i.e., patterns for collaboration practices, into the scoping process to overcome the detected problems.

The combination of scoping practices with collaborative patterns and thinkLets would improve the interaction of the participants, pracitcas that indicate how to brainstorm at the time of proposing the features and products that will be part of a product line will allow the different knowledge of the participants add possibilities, as

well as tinklets related to evaluative patterns allow the participants to contribute different visions in the evaluation of features and this facilitates the decision making as opposed to which they will be part of the common features or the variability of the line. In a first version of our proposal, we have considered associating 6 thinkLets of 6 collaborative patterns.

The paper is organized as follows. Section 2 presents background. Section 3 corresponds to scoping exploratory study the section 4 includes the study findings in the session 5, the thinkLets are proposed for product line software scoping Finally, Section 6 describes conclusions and further work.

2 BACKGROUND

There are many authors mentioning the importance of collaborative practices at defining the scope for a software product line. This section presents a background on existing methods for scoping considering collaboration, the ThinkLet collaboration patterns proposed by many authors and the Small-SPL process used in our study,

2.1 Thinklet collaboration patterns

Briggs et al. [4][13] proposed *Collaboration Engineering* as an approach for designing tasks where collaboration is recurrent and provides high-value, i.e., tasks where multiple individuals combine their knowledge and efforts to achieve a mutual goal [4]. According to these authors, there are a few *collaboration patterns* that characterize how "group's activities" can be transformed into "team's activities" [4]. The following are these collaboration patterns [4][13][19]:

Diverge: Move from having fewer to having more concepts shared by the group.

Reduce: Move from having many concepts to a focus on fewer concepts than group deems key elements

Clarify: Move from having less to having more shared understanding of concepts and of the words and phrases used to express them, and to assure that team members agree and understand the meaning of concepts, and of the relationships among ideas the group is considered.

Organize: Move from less to more understanding of concepts and relationships among concepts the group is considering and achieve organizing the concepts into categories.

Evaluate: evaluate the relative value of the concepts with respect to one or more criteria.

Build consensus: Move from having fewer to having more group members who are willing to commit to a proposal.

To design (or re-design) a process, Collaboration Engineering proposes design patterns, named *thinkLets* [14], to design tasks considering the above collaboration patterns. Each thinkLet is an scripted technique, which specification describes inputs, steps and outputs. Process designers analyze the process to improve, determine the most appropriated thinkLets and use them as building blocks [19] to define predictable and repeatable tasks involving "people working together toward a goal." [13].

2.2 Small-SPL process

Small-SPL is a process for SPL engineering for small development companies based on the SEI's framework [5]. The life cycle of Small

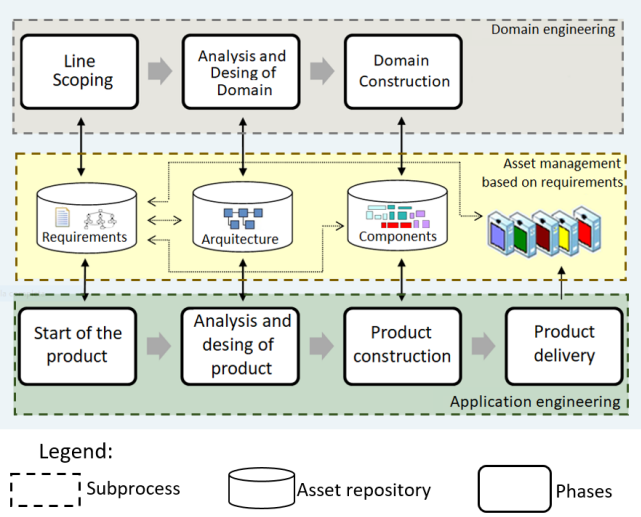


Figure 1: Small SPL process

SPL includes three subprocesses: Domain Engineering and Product Engineering, geared by a third subprocess called Asset Management based on Requirements. The domain engineering responsible for the production of the assets, the domain engineering of the construction of products, and the Asset Management based on Requirements allows to communicate the two fundamental processes in the construction of a line of processes, facilitating the identification, development , documentation, storage, search and use of the assets that will be used in the development of the products of the line. See Figure 1 [5]. It starts with *Line Scoping*, an activity that defines the products that comprise a software product line, specifies the features and requirements for each product and sketches the reusable assets that must be developed.

Figure 2 describes the scoping tasks. It starts studying the domain of applications for the product line and identifying the needs of the stakeholders. Then, the process continues exploring existing solutions in the same domain, listing possible products for the product line and identifying the features for each product. Once, the involved stakeholders have achieved agreements regarding products and features, the process follows establishing common and variable features and diagramming a feature model for the product line.

Note that, nowadays, the scoping in Small-SPL does not define its collaboration tasks using thinkLet patterns.

2.3 Product Line Scoping considering collaborative practices

There are many proposals for SPL scoping, i.e., for determining the products that make up the line, delimiting the subdomains and sketching the reusable assets [16][17][21][27]. All of them highlight the importance of involving participants with diverse knowledge and expertise [16][21], However, most of these approaches only name the roles or provide brief descriptions [22][26]. A few detail

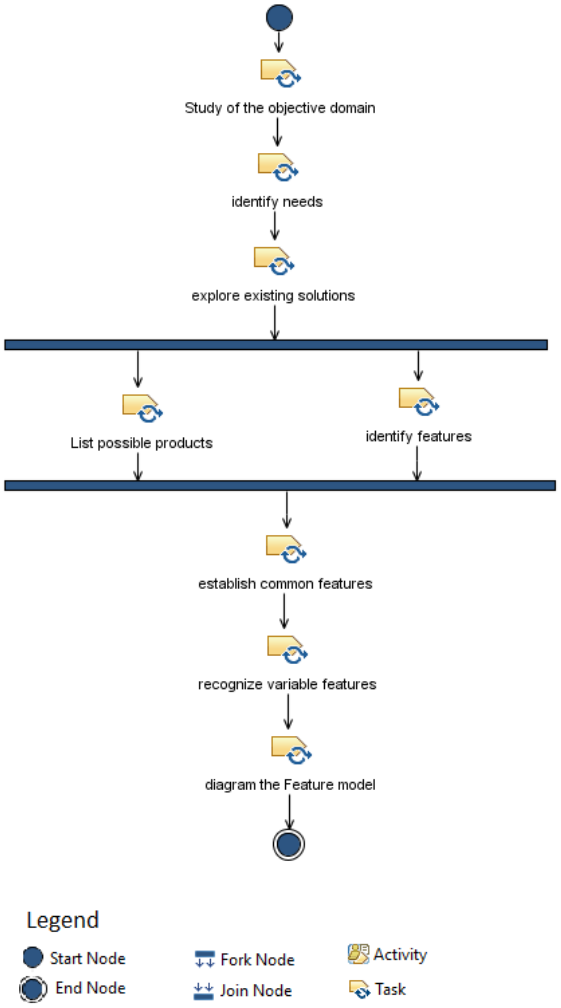


Figure 2: the task Scoping in the process Small SPL

the tasks in which these stakeholders must participate. the interdependencies among them [22][26] or the impact of human factors to the correct definition of the scope [18].

Some proposals have analyzed the effect of communicative factors on SPL scoping:

Helferich et al. [15] described differences on the marketing and engineering perspectives. According to them, while the marketing perspective focuses on the usage of the products, their sales channels, and the targeted market segments, the SPL Engineering focuses mainly on aspects like time-to-market, costs, or platform development. The scoping must be a mapping function between marketing and engineering aspects.

Rommes [26] states that SPL scoping is more a communication problem than an economic one. There, the challenge is to achieve that people with different needs, concerns and priorities can participate and cooperate to bound the scope. He proposes user scenarios that can be sketched and selected in a collective manner using the natural language, claiming that all the stakeholders might understand it.

Carbon et al. [7] proposes "Product Line Planning Game", a strategy that uses "stories of reuse" as a mean for obtaining feedback of application engineers and domain engineers.

Noor et al. [22][24] proposed the "Collaborative approach for product line scoping", an approach that considers the already developed products by the company, aiming to obtain a balance between business aspects and technical concerns. This approach focuses on the Product Portfolio Definition, the legacy products and the experience acquired developing these products.

The RiPLE-SC method [3] combines scoping practices with agile methodologies. It includes a detailed definition of the roles, its responsibilities and the tasks in which they participate. In addition, it describes agile practices regarding the interaction among participants. The authors, in a subsequent industrial case study [12], evidenced problems regarding inefficiencies of communication, interaction and cooperation among participants. They considered that using collaborative engineering patterns could be an opportunity to support stakeholder participation.

RiPLE-ASC [11] is a scrum-inspired process for SPL scoping. The proposal seeks to improve adaptability and feedback of the scoping process and define the scope in an incremental way, improving feedback among stakeholders. The approach suggests the use of collaboration patterns in three of its five activities; however, it does not give detailed information about how collaborative tasks are performed.

In summary, although there are many proposals considering multiple stakeholders and their interactions, only a few define collaboration practices: RiPLE-SC has evidenced the need for defining properly these practices and RiPLE-ASC uses collaboration patterns to specify some of them.

3 AN EXPLORATORY STUDY ON SCOPING

We are interested on exploring the problems related to collaboration practices for scoping product lines and the thinkLets that can be applied to improve the process. This section describes an exploratory study, The study aims to give us empirical and exploratory knowledge about scoping task, regarding contributions, communication, and interactions of the participants in this task.

3.1 Research Questions

We posed three research questions:

RQ1: What problems related to the interaction and collaboration of the stakeholders can be observed when groups of developers define a scope for a Software Product Line?

RQ2: What thinkLets can be used to overcome the collaboration tasks for scoping a Software Product Line?

RQ3: How can a process, such as the Small-SPL, be improved by using the identified thinkLets?

3.2 Experimental Design

We performed a exploratory study to find answers to our research questions. In this study, several groups of people developed a complete software product line after defining a scope with diverse

stakeholders from a real company. After each software product line was developed, we performed multiple evaluations and discussions with each group to determine problems and identify tasks that can be improved.

Participants. We carried out our experience in an academic environment. On the one hand, the developers were 24 students of a elective course in the third year of the Informatics Engineering at Universidad Colegio Mayor del Cauca (Colombia). On the other hand, the stakeholders or line customers were staff members of four organization units of METREX S.A.[1], a company that manufactures and commercializes flow meters. The participants' selection strategy was by availability and its members were selected at random [28].

The students were divided into development teams by organization unit, were distributed as follows: for Production unit 9 students with 4 products, laboratory 6 students with 3 products, warehouse 6 students with 3 products, and human talent unit 4 students with 2 products. The boss of each unit was in charge of interacting with the development group. Meetings were initially scheduled every 3 weeks.

The participants of the company were, the head of each unit and some units employees. During the first meeting where the students knew the processes and procedures of each unit touring the facilities, the bosses determined the operators who participated in this capacitation; for example in the production unit, some of the operators They explained their functions when the students went through their job positions. In some meetings, the boss of the production unit and laboratory boss were accompanied by one of their operators, also the bosses appointed one or two employees to replace in some meetings when they could not attend because of their priority work commitments.

Product Line. Students developed a product line of microgames for training employees of the company. The products line were questions and answer games related to the topics of four company departments, production unit, laboratory, warehouse, and human talent unit.

The product line was composed of 12 games, distributed so: 4 for the production unit, 3 for laboratory, 3 for warehouse, and finally 2 games for human talent unit.

The line of training games shared the design basis on questions of the unit's procedures and the possible answers, but also each unit has its own requests that make the games different. One of the departments emphasized in its production process for which the boss requested to include videos or animations previous to the game as a training. While the objective of the warehouse is the recognition of the pieces of the counters, whereby requires the use of the images of these pieces in the games of this unit.

Tasks performed. The students were organized in many development groups, each one working with staff members of an organization unit of the company. The project comprised activities for training the developers, defining the scope for the product line and developing the diverse products of the line.

[1]https://www.metrex.com.co

Training. The development teams did not have previous experience in serious games neither in SPL development. We started the process with two training paths:

SPL and serious games A 15-hours training on development of SPLs and serious games was carried out at the beginning of the project by the lecturer in charge of the course.

Game development In addition, a 32 hours training in game development with Unity was taught by two experts in game development. This training on game development was carried out gradually and in parallel to the progress of the project.

Scoping. After the initial training on SPL, each development group started to define the scope following the Small-SPL process. For this task, developers meet with staff members of the company during two hours each 15 days. As mentioned before, four organization units participated in these meetings.

The first meeting was held among the group of students and flow meter company persons. In this meeting, the chief of staff was in charge of communicating general aspects company, its organizational structure and some remarks about its operation. She explained the objective of the project from the point of view of the company, described the participating departments and the products per department and presented the heads of each department who would be the ones who would interact with the development groups.

A second meeting was held between department heads and development teams, where the particularities of the operative processes of each department and the training needs of its operators were explained. This meeting last two hours.

The students held a workshop with the objective to propose possible products and their features, as well as the groups identified the common features and the variability between the products; The proposals were presented and validated in a fourth section with the representatives of the departments.

Although we initially planned only four meetings, other two scoping meetings were necessary because (1) there were differences between the product proposed by the developers and the expected by the company and (1) changes on the staff members that participated in the meetings. Finally, a scope refining session was carried out in which the students participated cooperatively to be able to agree on the characteristics and the type of products that conform the line. The Figure 3 shows photos of the scoping meetings.

During the four months that the project lasted, a constant follow-up was done to the development group, el teacher accompanied them to all the meetings with the boss of the units, and meetings were held every two weeks with the whole group and with each subgroup. La recollection of study information was made during all the meetings by means interviews, and recordings, also through artifacts produced such as models and prototypes, The teacher was also in constant contact with the chief of staff of the company to monitor the vision of the project and company interest

Scoping was performed in four steps. First, there was a general meeting in which the boss of the units participated, in which the chief of staff explained what was the objective of the project, the process of training the employees and that it was intended to include serious games and in that At the moment they were looking to apply,

Figure 3: Scoping for a SPL of training games

then the group was divided according to the needs of each unit, and these subgroups met with each boss to understand the processes and specific procedures, such as the expectations of each unit boss Second, the developers proposed the training products and features for each organization unit. They presented and discussed these proposals at a meeting with the coordinator of the corresponding office. Third, a task for refining the scope was carried out with the cooperatively participation of all the students. Finally, product proposals were presented to the heads of department and a vote was taken on the features of the products.

Development. The study was an elective course in which the students had classes once a week with a duration of 4 hours; Meetings were scheduled every 15 days with the company, of which 2 of the planned ones were canceled. and the intermediate weeks meetings were held in the class, in which group meetings were held to review progress and agree on tasks, meetings were also held with each sub-group to verify the progress of each product, and then meeting with experts in unity for counseling; During all the time the development of the project was observed, and the documents and advances of the software assets were received.

Evaluation and Analysis. The teacher accompanied the students in all the meetings, interviews and workshop made with the bosses of units and project leader of the company, with the objective of carrying out the observations of the study.

The bosses of units made the evaluation of the games during the delivery of the games, where each of the heads of the department evaluated the micro-games, in relevance to the objectives of the project, correspondence with the requests, functionality and usability.

A printed questionnaire that was completed individually by each unit boss, during the presentation of the games, this form contained 12 items related to the correspondence with the requests, functionality, and usability of the games developed, to was evaluated with on a scale of 1 to 5, where 1 implied the minimum degree of satisfaction and 5 the maximum.

4 FINDINGS OF OUR STUDY

We evaluate the advances of each development team during all the process. Here we describe the findings obtained after defining a scope for the product line and after developing the products. In addition, we present some findings regarding the Small-SPL process.

4.1 Findings after scoping the product line

After defining the scope for the product line, the following observations were made:

- The scoping activity in Small-SPL process is not detailed enough and specific enough to be easily followed by the teams and that they can deduce concrete steps to follow. This situation was evidenced in the observations made during the development of the study and the questions asked by the members of the development group y doubts about the scoping artifacts that should be done. These observations are not exclusive of scoping activity in the Small-SPL process, but also of other approaches to scoping and that have been reported in other articles such as industrial case study reported by da Silva [12].
- The participation of the company members were limited. In addition, there were few changes in the participating people. However, these changes were made minutes before the meetings and some of the new people were not aware of the project. This made difficult the scope definition and, regretfully, some teams felt as they started again the task of identifying features and products at each meeting.
- Among the departments of the company, there were some different interests in the project and they expected different products for training games. The lack of knowledge and coordination of project hinder to identify the products and features that make up the training micro-games line.
- The application of the collaborative practices in the scope refinement task allowed that the developers agreed on products, prioritize features and unify the used terminology, also allowed unifying the points requested by the four bosses of the units; these agreements were not achieved in previous tasks, tasks that did not apply collaborative techniques.

4.2 Findings after developing the product line

At the end of the project, the teams developed thirteen products, However, ten corresponded to the line, three products did not belong to the line, and two products that should have been included were excluded. The products out of the line did not satisfy the expectations of the unit boss and they represented losses in effort and time.

We interviewed the development teams to identify problems and failures in the communication among the participants that affected the scoping. The following are some observations:

- The lack (or low-level) of communication between developers and department coordinators (the domain experts) made difficult to identify the features and products to should develop
- There was not enough cooperation among developers, analysts and the software architect. This was evidenced by failures in the development of the assets that were made

as individual components and not as reusable assets, evidencing drawbacks in the identification of assets and in the production plan.
- A lack of communication between the project manager of the company and the department coordinators was also noticeable. This made it difficult to identify and fulfill the objectives of the line, as well as to prevent the features and products to be developed.

This paper reports our exploratory study, it evidenced problems related to the lack of communication between the developers and domain experts and, additionally, poor cooperation between developers, analysts, and architects. Also, was identified as a lack of communication between the company chief of staff and the department coordinators. Among the departments of the company, there were differences between the conceptualization of products and their type games or evaluative application, and also in the construction of products. The lack of experience of the development group in SPL made it difficult for it to identify and produce common assets.

The application of the collaboration patterns in the specific tasks allowed to observe the differences in communication, collaboration, as well the level of acceptance and concordance of results obtained, between the tasks in which they were applied and those that were not.

4.3 Findings regarding Small-SPL

Finally, we must mention some problems regarding our specification of Small-SPL: The activity of scoping is focused on the product portfolio, does not include tasks for scoping the domain or the assets, It does not include templates for the artifacts that make up the scope neither provide help or guidance for development groups producing them.

4.4 Threats to Validity

There are some threats to validity of the study. It was analyzed SPL scoping in an academic domain that although it does not allow to generalize the observations to other domains, the results can be compared with those obtained by other studies, and the common points serve as a basis to propose a method that seeks to improve communication and collaboration in scoping. Hence, it needs to perform evaluations of the method to be proposed in other domains, academics with experience in SPL and software companies' domains.

5 THINKLETS FOR PRODUCT LINE SCOPING

We have identified some thinkLets that can be used to improve the process proposed by Small-SPL for scoping product lines. Following the guidelines for Designing Collaboration Processes [19], we analyzed the diverse tasks in the process and determined the most appropriated thinklets.

For the selection of which thinkLets must be used in every scoping task. First, we done was an analysis of each task: how many roles are involved in the task, what type of interactions involves the task like generate ideas, arrange them, organize them, classify them or evaluate them, the number of interventions that are expected, the inputs and outcomes of task.

Table 1: Scoping Tasks with Collaborative Patterns and Thinklets

Task	Collaborative Pattern / Thinklet	Purpose
Identify product line goals	Diverge / OnePage Reduce / FastFocus	Participants agree on the objectives of the line
Identify domains	Diverge / OnePage Reduce/ FastFocus	Participants select the target subdomains of the product line
Identify features	Diverge / OnePage Reduce / Pin the Tail on the Donkey	Participants exchange ideas of possible features that products may have
Identify products	Diverge / OnePage Reduce / FastFocus	Participants exchange ideas on possible products
Connect features, products and domains	Convergence/ Broomwagon	Select from the list of features those that correspond to subdomains and products
Describe domains	Build consensus/ StrawPoll	Agree on the concepts and characteristics of the identified subdomains
Specify product feature Matrix	Organize/ Popcorn-Sort	Classify the features of each product
Products and features assessment	Evaluate/ Bucket-wall	Approve the features of the products, by voting of each participant

Thinklets were selected according to the most suitable collaborative pattern according to the type of interactions involves the task and its objective in order to find the best fit the collaborative activity. Each collaborative pattern has different associated thinklets. For the selection of the thinkelts, a first inspection of the tasks was carried out, the number of participants, the entries and outcomes. This is a first version of the method, it is necessary to make a revision of the characteristics of each task and the selection of the thinkltes.

Putting a group of people around a task as scoping does not guarantee a real collaboration. In that way it is necessary to structure activities convey collaboration [8]. We search propose a method for product line scoping that incorporates collaborative elements with the objective of facilitating communication and cooperation among the participants in the building of the artifacts that compose the scope.

This proposal is a first approach to the integration of collaborative elements such as patterns and thinkLets in the tasks of scoping, The idea is to direct the steps of the tasks towards the construction of the artifacts that make up the scope. For which it is desired to establish which are the artifacts that make up the scope, and which are useful for the following phases of development of a product line, and establish the traceability between these artifacts and the steps specified in the tasks applying the collaborative patterns

6 A REVISITED PROCESS FOR SPL SCOPING

We have defined a a first version of a collaborative method for SPL scoping, considering the detected problems and the thinklets that can be used to overcome them.

we opted for defining a method considering the three levels of scope, whereby is necessary, reviewing other SPL scoping approaches instead of just adding a few tasks to Small-SPL,

Nor do we ignore the previous works, approaches that have considered communicative aspects such as a collaborative approach for product line scoping [23][24], and an agile scoping process for SPL [2] and A People Oriented Approach to Product Line Scoping [26].

As we also consider important to study other approaches that help us identify the components of scoping activity: Practical Guide to Product Line Scoping [17], and the unified approach for the SPL Scoping [20].

Following practices for method engineering [25], we wait to identify *method components* from inputs, outputs, roles and the description of its steps from the different base approaches.

In this first version, the identified method components corresponded to tasks. Each task was combined with thinkLets that were considered appropriate to the objective and particularities of the task. the technique described by the thinkLet was applied to the description of the steps that should be performed in the task.

In future versions of the method, it is expected that task guidelines allow the deduction and realization of concrete steps related to artefact (outcome) to be built in each task or method component [19]..

Roles. We have identified five roles for the scoping process [17]:

scoping expert the person that drives and customizes the scoping activities and conduct workshops and interviews.

marketing expert who provides their knowledge about the products and the application domain.

domain experts who provide their knowledge concern of application domain.

technical experts who give technical aspects of development products.

SPL manager who is responsible for routing development of activities for the development of SPL.

Please note that, in our process, the scoping expert is responsible for guiding the execution of the tasks collaboratively applying the selected thinklets.

Tasks. Figure 4 shows the proposed tasks for product line scoping. The process starts by studying the general domain, identifying goals and identifying domains of the product line.

First, the method defines the target domain then, the process follows by identifying the products for the line and the featured for each product. After, the next task connects the features of products and domains using an organizational pattern. Later, the process describes the domains and specify the line using product feature matrices, Finally, the process ends by assessing the products and features, identifying the assets to develop and establishing the production plan.

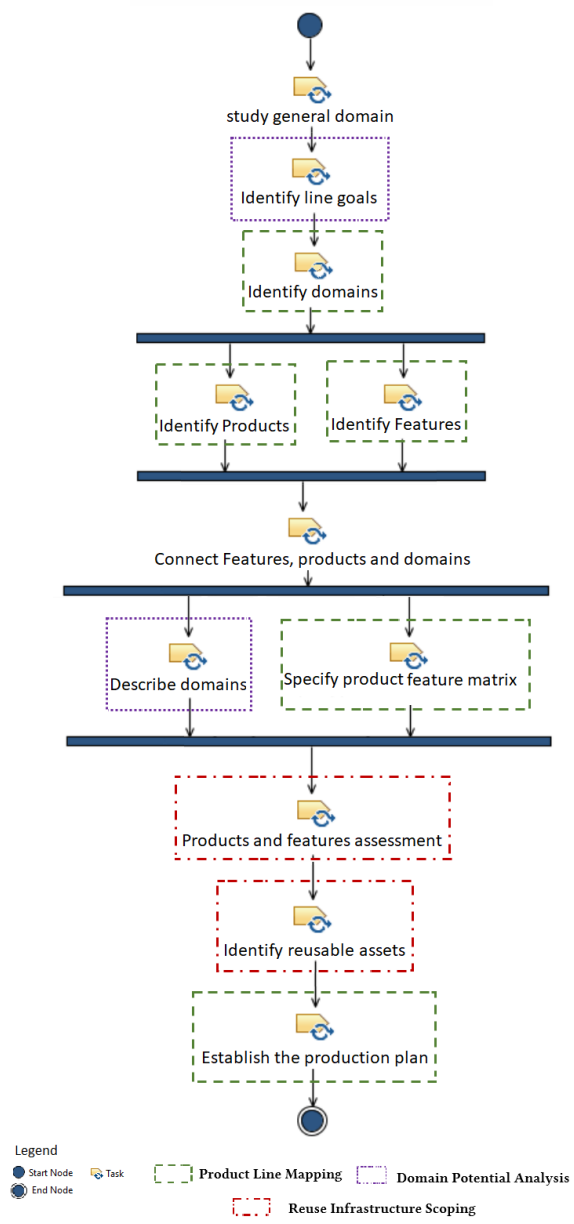


Figure 4: Relationship between the scoping tasks with the general collaborative patterns

7 CONCLUSIONS, LIMITATIONS AND FURTHER WORK

Although, scoping a SPL requires the participation of different interdisciplinary roles that contribute from different areas of knowledge and expertise, achieve effective participation in this activity requires of a systematic and detailed process that guides the team towards useful artifacts. Our project moves towards a new perspective of the SPL scope activity and visualizes scoping in a collaborative way,

then we must identify what collaborative pattern and thinkLets can be applied.

For propose the scoping activity collaborative way, first, we should identify the tasks that need to be realized by collaboratively way, then the patterns and thinklets that can be used to achieve effective participation and looking that the task is easy and understandable but without increased the effort required

This article presents a first approach where some patterns and thinkLets were used in some tasks and the effect was evaluated, after which a first version of the proposal is presented examining the scoping tasks and making a first combination with the collaborative patterns.

The study was designed to achieve an objective understanding of the problems and the effect of the Thinklets on teamwork, nevertheless, the results depend on cultural and organizational aspects (size, type, business model) that were part of the experience. Therefore, more empirical evidence is required to prove that the findings presented here are replicable in other contexts defining a SPL scope.

As a future work, we want to build a collaborative method based on existent methods considering reflections of Rommes [26], Noor [24], Carbon [7], and different experiences in order to include collaboration aspects in a systematic and practical way to encourage the participation and interaction of the roles involved in order to determine a useful SPL scope. This requires to organize and refine the process according to practical facts and validate the new method in industrial settings.

ACKNOWLEDGMENTS

This work was partially funded by the project "Network of training of human talent for social and productive innovation in the department of Cauca (Innovacción) executed by the Universidad del Cauca under code 3848, by the Research Vice-rectory of the University of Cauca, the Institución Universitaria Colegio Mayor del Cauca and Universidad de los Andes. We thank METREX S.A. for your participation in this project.

REFERENCES

[1] Faheem Ahmed and Luiz Fernando Capretz. 2010. An organizational maturity model of software product line engineering. *Software Quality Journal* 18 (2010), 195–225.

[2] M.a Balbino, E.S.b c De Almeida, and S.a c Meira. 2011. An agile scoping process for Software Product Lines. *SEKE 2011 - Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering* (2011), 717–722.

[3] Marcela Balbino Santos de Moraes, Eduardo Santana de Almeida, and Silvio Meira. 2011. An Agile Scoping Process for Software Product Lines. In *16th International Software Product Line COnference (SPLC'12)*. ACM, Salvador, Brazil, 225–228.

[4] Robert Briggs, Gwendolyn Kolfschoten, Vreede Gert-Jan, and Dean Douglas. 2006. Defining Key Concepts for Collaboration Engineering. In *12th Americas Conference On Information Systems, AMCIS 2006*. Association for Information Systems, AIS, Acapulco, Mexico, 117–124.

[5] Marta Cecilia Camacho Ojeda and Julio Ariel Hurtado Alegría. 2013. *SPL en las PYMES desarrolladoras de software del Cauca: una experiencia desde Colmayor*. Master's thesis. Universidad del Cauca, Popayán, Colombia.

[6] Luiz Fernando Capretz and Faheem Ahmed. 2009. *Software Product Line Engineering: Future Research Directions.* Nova PUblishers, NY, USA, 69–92.

[7] Ralf Carbon, Jens Knodel, and Dirk Muthig. 2008. Providing Feedback from Application to Family Engineering. In *12th International Software Product Line Conference (SPLC 2008)*. IEEE, Limeric, Ireland, 180–189.

[8] cesar a collazos, luis a guerrero, jose a pino, stefano renzi, jane klobas, manuel ortega, miguel a redondo, and crescencio bravo. 2007. Evaluating collaborative learning processes using system-based measurement. *Educational Technology and Society* 10, May (2007), 257–274. https://doi.org/10.1126/science.1168450

[9] Paul Clements and Linda Northrop. 2001. *Software Product Lines, Practices and Patterns*. Vol. 3 ed. Addison-Wesley Professional. Part of the SEI Series in Software Engineering, USA.

[10] Paul C Clements. 2002. On the Importance of Product Line Scope, Frank Van der Linden (Ed.), Vol. 2290. Springer, Berlin, Heidelberg, 70–78.

[11] Ivonei Freitas da Silva. 2012. An agile approach for software product lines scoping. In *16th International Software Product Line Conference (SPLC'12)*, Vol. 2. ACM, Salvador, Brazil, 225–228.

[12] Ivonei Freitas da Silva, Paulo Anselmo da Mota Silveira Neto, Pádraig O'Leary, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. 2014. Software product line scoping and requirements engineering in a small and medium-sized enterprise: An industrial case study. *Journal of Systems and Software* 88 (2014), 189–206.

[13] Gert Jan de Vreede, Robert Briggs, and Anne Massey. 2009. Collaboration Engineering: Foundations and Opportunities. *Journal of the Association for Information Systems* 10 (2009), 121–137.

[14] Gert-Jan De Vreede, Gwendolyn Kolfschoten, and Robert Briggs. 2006. ThinkLets: a collaboration engineering pattern language. *International Journal of Computer Applications in Technology* 25 (2006), 140–154.

[15] Andreas Helferich, Klaus Schmid, and Georg Herzwurm. 2006. Reconciling Marketed and Engineered Software Product Lines. In *10th International Software Product Line Conference (SPLC'06)*. IEEE, Baltimore, USA, 23–30.

[16] Isabel John and Michael Eisenbarth. 2009. A Decade of Scoping – A Survey. In *13th International Software Product Line Conference (SPLC'09)*. ACM, San Francisco, CA, USA, 31–40.

[17] Isabel John, Jens Knodel, Theresa Lehner, and Dirk Mut. 2006. A practical guide to product line scoping. In *10th International Software Product Line Conference (SPL'06)*. IEEE, Baltimore, MD, USA, 3–12.

[18] Michael John, Frank Maurer, and Bjørnar Tessem. 2005. Human and Social Factors of Software Engineering: Workshop Summary. *SIGSOFT Software Engineering Notes* 30, 4 (2005), 1–6.

[19] Gwendolyn Kolfschoten and Gert-Jan de Vreede. 2007. The Collaboration Engineering Approach for Designing Collaboration Processes. *Groupware: Design, Implementation, and Use CRIWG 2007. Lecture Notes in Computer Science* 4715 (2007), 95–110.

[20] JIHYUN LEE, SUNGWON KANG, and DANHYUNG LEE. 2010. a Comparison of Software Product Line Scoping Approaches. *International Journal of Software Engineering and Knowledge Engineering* 20, 05 (2010), 637–663. https://doi.org/10.1142/S021819401000489X

[21] Marcela Moraes, Eduardo Almeida, and Silvio Meira. 2009. A Systematic Review on Software Product Lines Scoping. In *6th Experimental Software Engineering Latin American Workshop (ESELAW 2009)*. Universidade Federal de Minas Gerais, Brasil, 63–72.

[22] Muhammad A Noor and Rick Rabiser. 2006. A collaborative approach for reengineering-based product line scoping.

[23] Muhammad A. Noor, Rick Rabiser, and Paul Grünbacher. 2008. Agile product line planning: A collaborative approach and a case study. *Journal of Systems and Software* 81, 6 (2008), 868–882. https://doi.org/10.1016/j.jss.2007.10.028

[24] Muhammad A Noor, Rick Rabiser, and Paul Grünbacher. 2007. A collaborative approach for product line scoping: a case study in collaboration engineering. In *25th conference on IASTED International Multi-Conference: Software Engineering (SE'07)*. ACM, Innsbruck, Austria, 216–223.

[25] Jolita Ralyté. 2004. Towards situational methods for information systems development: engineering reusable method chunks. *Proceedings of the International Conference on Information Systems Development (ISD'04)* July (2004), 271–282.

[26] Eelco Rommes. 2003. A People Oriented Approach to Product Line Scoping. In *International Workshop on Product Line Engineering (PLEES'03)*. Fraunhofer IESE, Kaiserslautern, Germany, 23–27.

[27] Klaus Schmid. 2000. Scoping Software Product Lines, An Analysis of an Emerging Technology. In *Software Product Lines: Experience and Research Directions*. kluwer academic publishers, Denver, CO, USA, 513–532.

[28] Claes Wohlin, Martin Höst, and Kennet Henningsson. 2003. Empirical Research Methods in Software Engineering. *Lecture Notes in Computer Science* 2765 (2003), 7–23.

Evaluation of the State-Constraint Transition Modelling Language: A Goal Question Metric Approach

Asmaa Achtaich*
aachtaich@gmail.com
CRI, Université Panthéon Sorbonne
Siweb, Univ. Mohammed V
Paris, France

Ounsa Roudies
roudies@emi.ac.ma
Siweb, Univ. Mohammed V
Rabat, Morocco

Nissrine Souissi
souissi@enim.ac.ma
ENSMR & Siweb, Univ. Mohammed V
Rabat, Morocco

Camille Salinesi
camille.salinesi@univ-paris1.fr
CRI, Université Panthéon Sorbonne
Paris, France

Raúl Mazo
raul.mazo@univ-paris1.fr
CRI, Université Panthéon Sorbonne
Paris, France
Lab-STICC, ENSTA Bretagne
Brest, France
GIDITIC, Universidad EAFIT
Medellín, Colombia

ABSTRACT

Self-adaptive systems (SAS) are exceptional systems, on account of their versatile composition, dynamic behavior and evolutive nature. Existing formal languages for the specification of SAS focus on adapting system elements to achieve a target goal, following specific rules, without much attention on the adaptation of requirements themselves. The State-Constraint Transition (SCT) modeling language enables the specification of dynamic requirements, both at the domain and application level, as a result of space or time variability. This language, evaluated in this paper, enables the specification of a variety of requirement types, for SASs from different domains, while generating a configuration, all configurations, and number of possible configurations, in milliseconds. This paper presents these results, namely; expressiveness, domain independence and scalability, from the viewpoint of designers and domain engineers, following a goal-question-metric approach. However, being primarily based on constraint programming (CP), the language suffers from drawbacks inherited from this paradigm, specifically time related requirements, like (e.g. order, frequency and staged requirements).

CCS CONCEPTS

• **Software and its engineering → Reusability**; *Specification languages*; *Empirical software validation*; Domain specific languages.

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342417

KEYWORDS

Dynamic software product lines, modeling language, IoT, state machine, constraint programming

ACM Reference Format:
Asmaa Achtaich, Ounsa Roudies, Nissrine Souissi, Camille Salinesi, and Raúl Mazo. 2019. Evaluation of the State-Constraint Transition Modelling Language: A Goal Question Metric Approach. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3307630.3342417

1 INTRODUCTION

Experience shows that the specification of Self-Adaptive Systems (SAS) involves a variety of requirements, specifically, variability related requirements. This last category guarantees configuration, reconfiguration, customization and extension properties to meet distinct and evolving needs. Managing variability is at the heart of software product line engineering. Therefore, the SPLE framework, revisited in [1], is an obvious basis to define the artifacts necessary for the specification of such systems. State-Constraint Transition (SCT) is a modeling language, specialized in the specification of SAS with dynamic requirements. Besides the fact that it allows the specification of variability requirements, the SCT language provides the necessary tools to specify a product line that's altogether dynamic in terms of requirements, as a result of dynamic artifacts (variability in time), or dynamic dependencies (variability in space). The SCT language extends constructs from the Finite State Machine (FSM) [10] paradigm to describe the dynamic behavior of SASs designed as Dynamic Software Product Line (DSPL) and adopts the constraint programming semantics to generate constraint satisfaction problems.

However, to prove its practicality, few questions arise. The first one raises the concern of expressiveness, the second, the scalability, and the third one the domain dependency of the language. Expressiveness refers to the ability of SCT to specify as many requirements of SASs as possible. Scalability translates to the capacity of SCT to

model real world large systems. And finally, domain independency determines the aptitude of SCT to be used in a variety of domains.

This paper presents an evaluation of the SCT modeling language from the point of view of expressiveness, scalability, and domain independency. To do so, a Goal-Question-Metric (GQM) approach was conducted. The approach starts by setting a goal for the paper, which is described in terms of questions, whose answers determine whether the goal was achieved or not. Each question can be answered in a measurable way, by being associated to a set of metrics. The results of the evaluation show that 89% of the requirements generated from the running example were met. The 11% remaining refer to time related requirements which are still a major limitation of our language, inherited from a limitation of constraint programming. Scalability was proven, by implementing the three systems ranging from 42 to 495 elements, with execution times within milliseconds and a co-variance equal to 1.34. Domain independence was proven by implementing the three cases. Again, time-related requirements were left out, decreasing the number of requirement successfully specified for all three cases to 84%.

The remainder of the paper is organized as follows. Section 2 presents the particularities of SAS, thus motivating the need for a new language, and presents the SCT language in a nutshell. Section 3 describes the GQM protocol followed in this paper, and introduces the example cases used for the evaluation. Sections 4, 5 and 6 respectively present our findings regarding expressiveness, scalability and, domain-independence. Section 7 presents related work just before concluding in Section 8.

2 BACKGROUND

With regards to their versatile composition, various users and evolutive nature, self-adaptive systems can be considered exceptional with complex and diverse requirements. Thus, the need for formal modeling language, that supports its specificity. This section confirms this observation and presents the main concepts of the SCT modeling language.

2.1 Self-adaptive systems

Self-adaptive systems (SAS) are composed under the construct that a variety of artifacts are adjusted, in order to best fit the requirements of final users, under various contextual circumstances. The specification of such systems can be achieved by the means of Dynamic Software Product Lines (DSPL), where a configuration is a product that corresponds best to application and adaptation requirements. Therefore, at runtime, and according to high level constraints expressed in the product line, elements are accordingly instantiated.

However, it has been observed that these high level requirements, are too, dynamic. In other words, and for instance, elements that are mandatory in a specific context, might become optional, or be part of a group cardinality instead, in a different context. This can be a result of the evolution of the SAS in space and time. Unfortunately, current DSPL notations like [19] do not provide the necessary means to specify variability at the domain, application and adaptation levels, along with the corresponding behavior expected from the SAS, with a powerful expressiveness, scalability, and generality.

2.2 Overview of the SCT language

State-Constraints Transition (SCT) is a language for the specification of self-adaptive systems, which vary in time and space. This language allows to dynamically specify the requirements that govern the structure and behavior of the SAS, in the light of changes in its context, and eventual evolutions.

Based on the theory of finite automata [10] an SCT model is an abstract construction of all SAS **states**, as well as **transitions** that (may) activate some of them, following the occurrence of an event. The latter is specified as a conditional expression composed from context or system variables. Each state describes in a complete and evolutive way, the requirements that must be satisfied by the SAS, and each transition is activated by an event or a condition, referring to a situation of the context.

A **composite state** is used to describe the high level behavior of the self-adaptive system, and high level requirements that are enforced in such a state. This type of state constraints the whole SAS. Then, as the decomposition of the main composite state moves towards **simple-constraint states**, low level adaptation rules start to take place, by constraining part of the SAS in order to answer more specific requirements. Furthermore, **concern levels** are enclosed in the composite states to introduce execution flows, which contain states that can run in parallel. Eventually, at a moment in time, a main composite state is active; this one dictates the high level constraints that should be respected, along with parts of its hierarchy, which add another layer of constraints to the SAS.

Constraints are at the heart of the SCT language. They are implemented at the level of simple-constraint states, and can apply various types of restrictions on SAS elements, including (a) requiring its presence or absence in a configuration, (b) restricting its values, in the case of numeric elements, (c) restraining the number of its instances, in the case of multi-instantiated elements, or (d) determining its impact on other elements of the system or its context, and vice versa.

Models are all translated into constraint problems [18], [13], therefore, it goes without saying that the requirements described in each state, are also specified by means of constraints. The metamodel illustrated in Figure 1 presents a global view of the SCT language.

3 PROTOCOL

In order to evaluate the SCT modeling language, in terms of expressiveness, scalability and domain independence, the paper uses the Goal-Question-Metric (GQM) approach [23]. A target goal is declared, and a set of research questions, whose answers determine, by the mean of metrics associated to them, the degree to which the goal is achieved. The goal, questions and metrics are defined in Table 1.

3.1 Goal Questions Metrics

Goal: Analyze the SCT modeling language, for the purpose of evaluating it, with respect to its expressiveness, scalability and domain independence, from the viewpoint domain engineers, in the context of SASs.

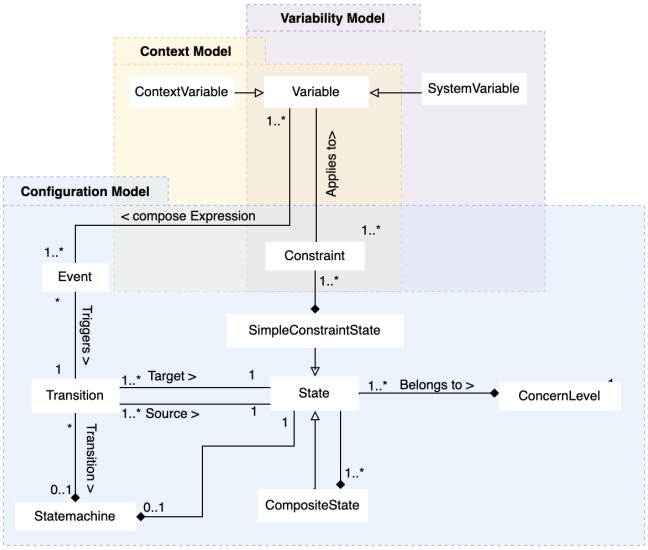


Figure 1: The SCT modeling language Meta-model.

Table 1: SCT evaluation questions and metric

Q1: Does the SCT language provide the necessary mechanisms to specify the requirements of Self-adaptive systems?
M1: Percentage of SAS requirements successfully specified by the concepts of the SCT language, with regards to the topology specified
Q2: How does the SCT language scale as the modeled systems get more substantial?
M2: execution times of systems of various scales **M3**: dependency ratio between execution times (ExT) and the # of elements
Q3: Can the SCT language specify self-adaptation requirements of systems from various domains?
M4: from all studied cases, the # of SAS from different domain applications that can successfully be specified using the SCT notation, with regards to the topology specified **M5**: from all cases, the # of requirements specified using the SCT notation

3.2 Cases

The evaluation of the SCT language was carried out using three cases of self-adaptive systems from three different domains. The Irrigation fleet is a case from the IoT field, the Gridstix case is from the Wireless Sensor Network (WSN) domain, and the landing Gear System case is an example of a complex system. The irrigation fleet is used to generate a requirement typology, which serves as a basis for the evaluation of expressiveness. All three cases are then implemented, first to demonstrate the scalability of the language, and to prove the domain independence. The number of cases implemented (i.e. three) is required to prove domain independency.

- **The irrigation fleet:** To maintain her field of Angelicas, Maria installed a fleet of devices to monitor and control the irrigation process. Her Smart irrigation system is a product derived from Greenlife Solutions, a company specialized in building user tailored irrigation systems. The system is composed of 2 Sprinklers, a Dripline and a Humidity Sensor. A year later, Maria decides to make of her plantation a small business. The accuracy and efficiency of the fleet, in a very dynamic environment, are therefore decisive for the outcome, the cost and the quality of the plantation. Along with 4 more double headed Sprinklers (Rotor and Spray), with adjustable rotations and pressures, 2 more Humidity Sensors, and a smart meter, new requirements have been expressed, for 2 different operating modes.
- **The Gridstix:** the Gridstix is a wireless sensor network (WSN) for detecting and predicting flooding. Each node of the WSN is equipped with sensors that detect the water depth and flow rate [11]. The communication between the nodes can be established using WiFi or Bluetooth protocols. The first one is more fault tolerant but requires more energy than the second one. Furthermore, the transmission can be set up using the shortest path or the fewest hops routing protocols. The first one is jeopardizing accuracy and fault tolerance, but consumes less energy. Finally, the data processing can be realized using centralized or distributed algorithms. Accuracy is achieved by the first type at the expense of energy efficiency. The system adapts these three aspects, depending on the state of the river, and battery level.
- **Landing Gear System:** The LGS is controlled digitally in nominal mode and analogically in emergency mode. In [4], the author did not consider the emergency mode, however, health parameters for all the equipment involved in the landing gear were elaborated, and are therefore, included in this implementation. Only normal mode requirements are fully specified and considered in this paper. The **pilot interface** is composed of the Handle, which can be switched to up or down, and three lights which inform on the position of the gears. The **hydraulic part** consists of (1) Three landing sets, each composed of a gear, a door, and latching boxes. (2) Six actuating hydraulic cylinders. (3) six electro valves, five of which set the pressure on the portion of the hydraulic circuit they are concerned with, and one general electro-valve which supplies the required electro-valve with power from the aircraft circuit. These electro-valves are activated by an electrical order, coming from the digital part. And finally, (4) A total of 54 sensors, which inform the digital part about the state of the equipment described above. The **digital part** is composed of two computing modules, which simultaneously execute the same software, to control gears and doors, detect anomaly, and keep the pilot informed of the overall state of the system.

4 EXPRESSIVENESS

The answer to Q1 depends upon a set of requirements deducted from the first case. This helps identify the needs of the users of SAS, and to define a typology that can be generalized for analogous systems. The typology comprises the following requirements:

Table 2: Mapping the irrigation fleet requirements to SCT concepts

Requirements	CV	SV	Cst	Ev	T	SCS	CS	CL	Example from SCT
Functional	-	-	-	-	-	-	-	✓	**concernLevel** Irrigation
Non-functional	-	-	-	-	-	✓	✓	✓	**simpleConstraintState** WaterEfficiencyMode **compositeState** WaterEfficiencyMode
Structural - Boolean	-	✓	-	-	-	-	-	-	**Boolean** Dripline
Structural - Numeric	-	✓	-	-	-	-	-	-	**Integer** Pressure [1,6] [0..60]
Structural - Domain	-	✓	-	-	-	-	-	-	**Integer Enum** Rotation [1,6] 0, 90, 180, 240, 360
Structural - Multi-instances	-	✓	-	-	-	-	-	-	**Boolean** Spray [1,6], **Boolean** Rotor [1,6]
Structural - Mandatory	-	★	✓	-	-	★	-	-	SIS = Sensor
Structural - Optional	-	★	✓	-	-	★	-	-	SIS >= Fertilizer
Structural - Group	-	★	✓	-	-	★	-	-	**simpleConstraintState** GlobalState (Irrigator ≥ Dripline) ⋀ (2*Irrigator ≥ Sprinkler[1]+Sprinkler[2]) ⋀ ((Dripline + (Sprinkler[1]+Sprinkler[2]))=Irrigator*1 ⋁ (Dripline + (Sprinkler[1]+Sprinkler[2])=Irrigator*2)) ⋀((Dripline * (Sprinkler[1]+Sprinkler[2])=0))
Structural - excludes	-	★	✓	-	-	★	-	-	Slave[1] >0 ⟶ Alarm[1]=0
Structural - requires	-	★	✓	-	-	★	-	-	Sprinkler[1] ≥ 1 ↔ Pressure[1] ≥ 1
Composition	✓	✓	✓	★	★	★	★	-	**simpleConstraintState** MaxEcoIrrigation Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+Sprinkler[4]+Sprinkler[5]+Sprinkler[6]<2
Contextual	✓	✓	✓	★	★	★	★	-	**simpleConstraintState** SurvivalIrrigation Dripline = 1
Parametric	✓	✓	✓	★	★	★	★	-	**Float Param** Consumption **Float Para**m OpConsumption **When** AccuracyMode **if** (PowerFailure = 0 ⋀ Consumption > MaxConsumption) **transition from** ...
Proportionality	✓	✓	✓	★	★	★	★	-	(temperature>5)⟶ HB[1]+HB[2]+HB[3]+ HB[4] +HB[5]=3
Optimization	✓	✓	✓	★	★	★	★	-	minimize(HB[1]+HB[2]+HB[3]+ HB[4]+HB[5])
Time	-	-	-	-	-	-	-	-	—-

- **Functional requirements** represent the functions of the system to be developed. In particular, the tasks and services expected from them. They are mainly action oriented [16] (e.g., the irrigation fleet shall regulate soil humidity).
- **Non-functional requirements** represent the expected properties and qualities of the system such as its performance, efficiency or service durability [6] (e.g., the irrigation fleet shall be water efficient).
- **Structural requirements** specify the constitution of a system (Boolean/numeric elements and multi-instances), specifying its common and variable components and the relationships that interconnect them (mandatory, optional, alternatives, group relation, excludes, requires) [3] (e.g., the irrigation fleet may use the Air conditioner to provide temperature regulation).
- **Composition requirements** specify the components involved in a configuration for a specific context, according to predefined rules. Composition requirements are a special case of structural requirements, concerned exclusively with physical elements [14] (e.g., if a power failure occurs, the irrigation fleet shall replace the sprinklers with the dripline).
- **Contextual requirements** represents the operational configuration required in a specific context [2] (e.g. When Water Efficiency mode is activated, if the battery level of a humidity sensor <50%, the irrigation fleet must activate slave mode).
- **Parametric requirements** contain unvalued expressions that can be instantiated with a specific value at runtime [9] (e.g. When the Consumption > MaxConsumption), the fleet shall use the Sprinklers with Rotor mode)
- Proportionality requirements: specify the rules that define logical relationships between various elements of a system [22] (e.g. The irrigation fleet shall select respectively (3, 4, 5) heating bulbs, if the field temperature (> 5,> 0, -5))
- **Optimization requirements** define a maximum or minimum required for the values or cardinalities of numeric or multi-instantiated components [21] (e.g. When (Power Failure = True), the irrigation fleet must minimize the number of Master humidity sensors)
- **Time requirements** determine the time, order, frequency or execution time in which some operation are performed [5] (e.g. On October 1st, at 7pm, the irrigation fleet shall deactivate the alarms and activate the bulbs)

Table 2 maps each type of requirements with a concept from the SCT language. The abbreviations in the columns of the table refer respectively to : Context Variable (CV), System Variable (SV), Constraint (Cst), Event (Ev), Transition (T), Simple-Constraint States (SCS), Composite State (CS) and ConcernLevel (CL).

The symbol ✓ refers to the concepts that enable the specification of a requirement. The symbol (⋆) pertains to complementary concepts that aid with the specification of a certain requirement. Finally, the symbol (-) refers to concepts that do relate to the specification of a requirement.

Functional requirements can be specified by the means of the concern level concept. This allows the specification of various states within each concern level, which can be executed in parallel, thus allowing the satisfaction of all required functional requirements. Non-functional requirements can be represented by various states (simple-constraint states or composite State), each guaranteeing a specific level of performance. Structural and composition requirements are represented by interconnecting various types of mono or multi-instantiated variables (Boolean, Integer, Floats or domain restricted) by constraining their values. Context and system variables are declared in their respective domains and are used in a constraint expression to express a hierarchical or traversal dependency. Operational requirements are represented by the means of events, guarding the transition from one source state to a target state. The events are logical expressions that use context or system variables. Parametric requirements are first enabled by the means of parameters, which are special kinds of variables. They are used to define parametric events, or parametric constraints within a simple-constraint states. Finally, proportionality and maximization requirements are specified using variables and constraints. They are defined in a simple-constraint states state. However, since all concepts are eventually translated to a constraint program, time requirements could not be specified as temporal constraints still constitute a major limitation in constraint programming.

In conclusion, M1=8/9 types of requirements collected can be specified using the SCT language. Therefore, to answer Q1, 89% of the requirements expressed for SAS can be specified using the SCT language.

5 SCALABILITY

As SAS evolve, the number of elements to take into consideration during the modeling process could rise to thousands and even millions of elements. Scalability is therefore a very prominent aspects, as it determines the capability of the SCT modeling language to handle real world systems. Scalability, as presented in the following, refers to the execution times needed to generate a configuration, while performing operations such as All configuration, Valid configuration and Number of configurations. When an optimization requirement is involved, a Best Configuration operation is also part of the execution results. To answer Q2, the three cases were implemented, using a MacBook Pro, 2,7 GHz Intel Core i5, 8 GB 1867 MHz DDR3. The first case counts 151 elements, and is executed in M2-1= 0.001443 sec, the second case is composed of 42 and is executed in M2-2=0.006042 sec. finally, the third case, which counts 495, finishes the execution in M2-3=0.019769 sec.

Table 3: Config. of the irrigation fleet in a context (Cn1)

System element	Generated configuration
Dripline	0
Sprinkler	[1, 1, 1, 1, 1, 1]
Rotor	[1, 1, 1, 1, 1, 1]
Head	[1, 1, 1, 1, 1, 1]
Spray	[0, 0, 0, 0, 0, 0]
Pressure	[40, 40, 30, 30, 30, 30]
Rotation	[180, 180, 90, 90, 90, 90]
HS	[1, 1, 1]
Master	[0, 1, 0]
Slave	[1, 0, 1]

We can deduce that SCT is indeed capable of maintaining a good level of performance, as models ranging from 42 to 495 elements were executed within milliseconds. The co-variance ratio calculated was equal M3=cov(#elements , ExT)=1.3, proving that the independence between the two variables.

6 DOMAIN INDEPENDENCY

To answer Q3, three cases were implemented. First the irrigation fleet, then the Gridstix, and finally, the landing gear system. The requirements from the three cases were successfully specified, unless they are, partially or completely, time related requirements (e.g. When efficiency mode is activated, between 7pm and 8 am, the fleet shall disable all alarms OR The switch is closed each time the Up/Down handle is moved by the pilot, and it remains closed 20 seconds); these requirements could not be fully specified due to limitation in the constraint programming paradigm. Issues related to staged configuration, -where the order in which the problem is solved is relevant-, were handled by introducing intermediate input files, as presented by the columns of the Table 5, and by duplicating intermediate changing variables (IN and OUT). Furthermore, scheduled reconfiguration, where actions occur at/before/after/during specific times, are handled by Boolean variables, which are supposedly true at/before/after/during the specified times.

Results from the SCT generated configurations confirm this affirmation, therefore M4=3/3. The following describes for each case, a hypothetical scenario, which corresponds to a context (Cni), along with the expected reconfiguration of the SAS, and the actual configuration generated from the SCT model.

6.1 Case 1: Irrigation fleet

Context (Cn1): The battery levels of the humidity sensors are respectively 50,61,33. The water consumption is below the optimal measure. The electricity is up and running.

Expected configuration: The fleet shall maintain the humidity sensors whose battery levels are above 60 on Master mode, otherwise, on slave mode. The fleet shall use all 6 Sprinklers with Rotor heads, the instances 1 and 2 with a 180◦ Rotation, and a Pressure 40Pa, and 3, 4, 5 and with a Rotation 90◦ and a Pressure of 30Pa. Finally, the fleet shall activate all alarms for master humidity sensors. Table 3 shows the resulted configuration for the context described above

Table 4: Config. of the Gridstix in a context (Cn2)

System element	Generated configuration
WiFi	1
Bluetooth	0
SPTopology	0
FHTopology	1
DistributedProcessing	1
SingleNodeProcessing	0

6.2 Case 2: Gridstrix

Context (Cn2) : The battery level is high. The state of the river is emergency.
Expected configuration: The Gridstix is expected to maximize the fault tolerance and accuracy, at the expense of energy efficiency. Therefore, the system shall activate the WiFi as a transmission technology, the FH topology as a network organization protocol, and the distributed processing algorithm to calculate the flow rate. Table 4 describes the results from the SCT model.

6.3 Case 3: Landing Gear System

Context (Cn2) : The handle is down, and no anomaly has been detected.
Expected configuration: The LGS is expected to function on nominal mode, and the extension sequence is supposed to take place. We consider each task within a sequence as a configuration. Therefore, to achieve the sequence, a series of reconfigurations are made. The result of each on is also the input of the next one. Therefore, the result of an input 1 (IN1) is a reconfiguration (reconfig1). The latter is also the input (IN2) for the next reconfiguration (reconfig2), and so one. The important events for each scenario of the sequence (INi) are placed after the symbol (→). The elements concerned by the reconfiguration are highlighted as (StateBeforeReconfig → *StateAfterReconfig*). Table 5 summarizes the resulting behavior of the landing gear system function on analogical mode, as generated by the SCT model.

To calculate the number of requirements successfully specified for the three cases, we break down the count for each one. First, for the 10 requirements expressed in the irrigation fleet case, 9 are fully specified, and 1 is partially specified as it describes a time constraint. Second, 17 of the requirements described in the GridStrix case are fully satisfied. Finally, amongst the 11 requirements that summarize the overall expected behavior of the landing gear system, in normal mode, 9 requirements were fully specified, and 2 were half specified, since they are time related. The 10 failure mode requirements however were all half specified, since they all describe the behavior when a timeout is reached. We give the fully specified requirement a score 1. However, requirements that depend on time have a score 0.5, since they are partially specified. Therefore, the total number of requirements specified are M5=41,5/48.

7 RELATED WORKS

The language evaluated in this paper is not the first attempt on a formal language for the specification of self-adaptive systems. Authors in [24] propose a language that unifies the description and specification of key architectural characteristics, from different perspectives, using a small number of formally specified modeling elements, and a set of relationships that guide their composition. The Relax [25] method focus of uncertainty issues, to satisfy, to a certain degree, the non-functional requirements of the SAS.

Contrary to these notations, the State-Constraint Transition language does not specify adaptation scenarios. The language rather translates the requirements into constraints, which are verified with every significant change in the context. The solver is then responsible for finding a valid reconfiguration, if necessary. In addition, while most approaches have focused of the system‘ adaptation to variable conditions, to achieve a target goal, limited attention has been paid to the adaptation of requirements themselves.

Modeling variability is a key aspect in this paper, and a popular topic in the last decade. As a matter of fact, several authors have proposed languages for the specification and the management of this variability. Djebbi et al. [7] proposed a constraint programming based generic language. Mazo et al [13] proposed a generic meta-model for the specification of variability. While Dumitrescu et al. [8] adopted concepts from the SysML notation along with variability concerns. However, these notations did not allow the specification of typical requirements of SASs.

To bridge the gap between these variability approaches and self-adaptation requirements, several language were based on established notations, extended with variability concepts. In that sense, a first attempt was made by Sawyer et al [20], by extending goal-based notations with the concepts of constraint programming from [7]. The notation was nevertheless considered rigid. A second attempt was made in by extending the meta-model proposed in [15] with SAS-specific concepts. The approach was considered more generic and extensible, with good expressiveness, but an ontological analysis has revealed that the concept of state, which is lacking in all previous notations, is essential to the specification of possible evolutions [17].

8 THRETS TO VALIDIY

In this paper, a modeling language for the specification of self-adaptive systems is evaluated. Three cases are used to asses expressiveness, scalability and domain independency.

The first case is drawn from a study of the irrigation domain, including literature documents, and exchange with domain specialists. The requirements drawn from the study may therefore be inconclusive. Consequently, expressiveness of the language is satisfactory, in the context of the set of requirements presented above. Furthermore, scalability was tested by calculating the covariance of times of execution between the three cases, whose maximum number of elements is 495. The results cannot be undeniably proved for all scales, but can be generalized to a certain extent.

9 CONCLUSION

In this paper, the State-Constraint Transition (SCT) modeling language is evaluated. SCT models are used for the specification of self-adaptive systems, as dynamic software product lines. They are specifically efficient to represent requirements variability, both in time and space. The language is based on the theory of finite state machines and constraint programming to define configuration

Table 5: Configuration of the landing gear system in a context (Cn3)

	Generated configuration						
Input/ouput Sequence task	IN-1	OUT-1/IN-2 *ReCfg1*	OUT-2/IN-3 *ReCfg2*	OUT-3/IN-4 *ReCfg3*	OUT-4/IN-5 *ReCfg4*/5	OUT-5/IN-6 *ReCfg6*	OUT-6/IN-7 *ReCfg7*
Handle	2	2	2	2	2	0	0
Handle-Sensor	→**[2,2,2]**	[2,2,2]	[2,2,2]	[2,2,2]	[2,2,2]	[2,2,2]	**[2,2,2]**
G-cylinders	1	1	1	2	2	2	2
D-cylinders	1	1	2	2	1	1	1
f-gearPosition-Sensor	[2,2,2]	[2,2,2]	[2,2,2]	→**[1,1,1]**	[1,1,1]	→**[1,1,1]**	[1,1,1]
r-gearPosition-Sensor	[2,2,2]	[2,2,2]	[2,2,2]	→**[1,1,1]**	[1,1,1]	→**[1,1,1]**	[1,1,1]
l-gearPosition-Sensor	[2,2,2]	[2,2,2]	[2,2,2]	→**[1,1,1]**	[1,1,1]	→**[1,1,1]**	[1,1,1]
f-door-Sensor	[1,1,1]	[1,1,1]	→**[2,2,2]**	[2,2,2]	→**[2,2,2]**	→**[1,1,1]**	[1,1,1]
r-door-Sensor	[1,1,1]	[1,1,1]	→**[2,2,2]**	[2,2,2]	→**[2,2,2]**	→**[1,1,1]**	[1,1,1]
l-door-Sensor	[1,1,1]	[1,1,1]	→**[2,2,2]**	[2,2,2]	→**[2,2,2]**	→**[1,1,1]**	[1,1,1]
Hin	[1,1,1,1,1]	[1,1,1,1,1]	[1,1,1,1,1]	[1,1,1,1,1]	[1,1,1,1,1]	[1,1,1,1,1]	[1,1,1,1,1]
Hout	[0,0,0,0,0]	[0,0,0,0,0]	→**[1,0,1,0,0]**	[1,0,1,0,1]	[1,1,0,0,0]	**[1,0,0,0,0]**→	**[0,0,0,0,0]**
General-EV	[0,0]→	→***[1,1]***	[1,1]	[1,1]	[1,1]	**[1,1]**→	***[0,0]***
close-EV	[0,0]	[0,0]	[0,0]	**[0,0]**→	***[1,1]***→	→***[0,0]***	[0,0]
open-EV	[0,0]	**[0,0]**→	***[1,1]***	[1,1]	***[0,0]***	→[0,0]	[0,0]
retract-EV	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	→[0,0]	[0,0]
extend-EV	[0,0]	[0,0]	**[0,0]**→	***[1,1]***→	***[0,0]***	→[0,0]	[0,0]
anomaly	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]
Time-to-close-switch	0	1	1	1	1	1	0

states; where states represent enforced requirements, which are translated into constraint, and transitions represent events that (may) drive the adaptations. This approach ensures a high level of expressiveness, and powerful reasoning [13].

A goal question metric approach is followed to evaluate the approach from three perspectives (effectiveness, scalability, and domain independence), using three different example cases (an irrigation fleet, a landing gear system, and a Gridstix case). First the expressiveness by deducing a requirement typology from the first case, and mapped them to the SCT concepts. The goal of this process is to answer the first metric, regarding the number of requirements that can successfully be specified by the modeling language. Second, scalability is investigated. To determine the capability of the language to support real world SASs, which can scale to thousands and even millions of elements, especially in the IoT realm. Finally, domain independent is confirmed. This quality proves that the language is not restricted to a specific domain, but can rather be used for a variety of systems, from different domains.

The results of the evaluation show first that the SCT language is expressive, as it can specify most elucidated requirements. Second, that representing SCT constructs as constraints, in a constraint satisfaction problem, is an efficient and scalable approach, since systems counting from 20 to 500 elements perform operations within milliseconds. Finally, that it can be applied for various domains, including IoT fleets of connected objects, WSNs, and complex systems.

While these results are satisfactory, the main issue of the SCT language remains related to time requirements. This limitation comes from the inability of constraint programming paradigm to support the various aspects of time. Some techniques were used to perform the tasks related to time requirements, like running the constraint program multiple times, similar to the works of [12], or by introducing Boolean time variables, managed by external processes. Therefore, however very common, adaptations that occur at specific times of the day, adaptations that take place in a certain order, or staged adaptations are outside the boundaries of the language, and clearly require more investigation.

ACKNOWLEDGMENTS

This work was supported by the Moroccan Ministre de lEnseignement Supérieur, de la Recherche Scientifique et de la Formation des Cadres, by the French Embassy in Morocco, and by the Institut Français du Maroc.

REFERENCES

[1] A. Achtaich, N. Souissi, R. Mazo, O. Roudies, and C. Salinesi. 2018. A DSPL Design Framework for SASs: A Smart Building Example. *EAI Endorsed Transactions on Smart Cities* 2, 8 (jun 2018), 154829. https://doi.org/10.4108/eai.26-6-2018.154829

[2] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastian Uchitel. 2009. Learning operational requirements from goal models. In *Proceedings of the 31st international conference on software engineering.*, IEEE Computer Society (Ed.). https://doi.org/10.1109/ICSE.2009.5070527

[3] V. Ambriola and V. Gervasi. 1998. Representing structural requirements in software architecture. *Systems Implementation* (1998), 114–127. https://doi.org/10.1007/978-0-387-35350-0_9

[4] Frédéric Boniol and Virginie Wiels. 2014. *Landing gear system.* Technical Report.

[5] Amedeo Cesta and Simone Fratini. 2008. The Timeline Representation Framework as a Planning and Scheduling Software Development Environment. In *Proc. of 27th Workshop of the UK Planning and Scheduling SIG.*

[6] Lawrence Chung, Julio Cesar, and Sampaio Prado Leite. 1999. *Non-functional requirements in software engineering* (vol 4 ed.). Springer Science & Business Media. 441 pages.

[7] Olfa Djebbi, Camille Salinesi, and Daniel Diaz. 2007. Deriving Product Line Requirements: the RED-PL Guidance Approach. In *14th Asia-Pacific Software Engineering Conference (APSEC'07).* IEEE, 494–501. https://doi.org/10.1109/ASPEC.2007.63

[8] Cosmin Dumitrescu, Raul Mazo, Camille Salinesi, and Alain Dauron. 2013. Bridging the Gap Between Product Lines and Systems Engineering : An experience in Variability Management for Automotive. In *17th International Software Product Line Conference (SPLC)*. https://doi.org/10.1145/2491627.2491655

[9] John Favaro, Hans-Peter de Hans, Rudolf Schreiner, and Xavier Olive. 2012. Next Generation Requirements Engineering Ocean Surveillance View project Thermal Analysis for Space View project. In *INCOSE International Symposium*. Vol. 22. No. 1. https://doi.org/10.1002/j.2334-5837.2012.tb01349.x

[10] David Harel. 1987. On the formal semantics of statecharts.pdf. In *2nd IEEE Symposium on Logic in Computer Science*.

[11] D. Hughes, P. Greenwood, G. Coulson, and G. Blair. 2006. GridStix: Supporting Flood Prediction using Embedded Hardware and Next Generation Grid Middleware. In *International Symposium on a World of Wireless, Mobile and Multimedia Networks(WoWMoM'06)*. IEEE, 621–626. https://doi.org/10.1109/WOWMOM.2006.49

[12] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 2 (2002), 256–290.

[13] Raúl Mazo, Camille Salinesi, Olfa Djebbi, Daniel Diaz, and Alberto Lora-Michiels. 2012. Constraints: the Heart of Domain and Application Engineering in the Product Lines Engineering Strategy. *International Journal of Information System Modeling and Design* 3, 2 (2012).

[14] Hokey Min. 1992. Selection of Software: The Analytic Hierarchy Process. *International Journal of Physical Distribution & Logistics Management* 22, 1 (jan 1992), 42–52. https://doi.org/10.1108/09600039210010388

[15] Juan C. Muñoz-Fernández, Gabriel Tamura, Mazo Raúl, and Camille Salinesi. 2014. Towards a Requirements Specification Multi- View Framework for Self-Adaptive Systems. *Computing Conference (CLEI), 2014 XL Latin American* 18, 2 (2014), 1–12. https://doi.org/10.13140/2.1.3132.8640

[16] Jane Radatz, Anne Geraci, and Freny Katki. 1990. IEEE standard glossary of software engineering terminology.

[17] Iris Reinhartz-Berger, Arnon Sturm, and Yair Wand. 2011. External Variability of Software: Classification and Ontological Foundations. Springer, Berlin, Heidelberg, 275–289. https://doi.org/10.1007/978-3-642-24606-7_21

[18] Camille Salinesi, Raul Mazo, Olfa Djebbi, Daniel Diaz, and Alberto Lora-Michiels. 2011. Constraints: The core of product line engineering. In *Fifth International Conference On Research Challenges In Information Science*. IEEE, 1–10. https://doi.org/10.1109/RCIS.2011.6006825

[19] Pete Sawyer, Raul Mazo, Daniel Diaz, Camille Salinesi, and Danny Hughes. 2012. Constraint Programming as a Means to Manage Configurations in Self-Adaptive Systems. *Special Issue in IEEE Computer Journal "Dynamic Software Product Lines"* 45 (2012), 56–63.

[20] Pete Sawyer, Raúl Mazo, Daniel Diaz, Camille Salinesi, and Danny Hughes. 2012. Using Constraint Programming to Manage Configurations in Self-Adaptive Systems. *Computer* 45, 10 (2012), 56–63. https://doi.org/10.1109/MC.2012.286

[21] Joseph Sifakis. 2015. System Design Automation: Challenges and Limitations. In *Proceedings of the IEEE*. 103(11), 2093–2103. https://doi.org/10.1109/JPROC.2015.2484060

[22] Earle Steinberg and H. Albert Napier. 1980. Optimal Multi-Level Lot Sizing for Requirements Planning Systems. *Management Science* 26, 12 (dec 1980), 1258–1271. https://doi.org/10.1287/mnsc.26.12.1258

[23] Rini van Solingen, Vic Basili, Gianluigi Caldiera, and H. Dieter Rombach. 2002. Goal Question Metric (GQM) Approach. In *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., Hoboken, NJ, USA. https://doi.org/10.1002/0471028959.sof142

[24] Danny Weyns, Sam Malek, and Jesper Andersson. 2012. FORMS: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems* 7, 1 (apr 2012), 1–61. https://doi.org/10.1145/2168260.2168268

[25] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. 2010. RELAX: a language to address uncertainty in self-adaptive systems requirement. *Requirements Engineering* 15, 2 (jun 2010), 177–196. https://doi.org/10.1007/s00766-010-0101-0

Accessibility Variability Model: The UTPL MOOC Case Study

Germania Rodriguez
Universidad Politécnica de Madrid
Universidad Técnica Particular de Loja
Loja, Ecuador
grrodriguez@utpl.edu.ec

Jennifer Pérez
Universidad Politécnica de Madrid
Madrid, Spain
jperez@etsisi.upm.es

David Benavides
ETSI Informática
Universidad de Sevilla
Sevilla, Spain
benavides@us.es

ABSTRACT

Several approaches to define Variability Models (VM) of non-functional requirements or quality attributes have been proposed. However, these approaches have focused on specific quality attributes rather than more general non-functional aspects established by standards such as ISO/IEC 25010 for software evaluation and quality. Thus, developing specific software products by selecting features and at the same time measuring the level of compliance with a standard/guideline is a challenge. In this work, we present the definition of an accessibility VM based on the web content accessibility guides (WCAG) 2.1 W3C recommendation, to obtain a quantitative measure to improve or construct specific SPL products that require to be accessibility-aware. This paper is specially focused on illustrating the experience of measuring the accessibility in a software product line (SPL) in order to check if it is viable measuring products and recommending improvements in terms of features before addressing the construction of accessibility-aware products. The adoption of the VM accessibility has been putted into practice through a pilot case study, the MOOC (Massive Open Online Course) initiative of the Universidad Técnica Particular de Loja. The conduction of this pilot case study has allowed us to illustrate how it is possible to model and measure the accessibility in SPL using accessibility VM, as well as to recommend accessibility configuration improvements for the construction of new or updated MOOC platforms.

CCS CONCEPTS

• **Software and its engineering → Software product lines**;

KEYWORDS

Software and its engineering, Software creation and management, Software development techniques, Reusability, Software product lines

ACM Reference format:
Germania Rodriguez, Jennifer Pérez, and David Benavides. 2019. Accessibility Variability Model: The UTPL MOOC Case Study. In *Proceedings of 23rd International Systems and Software Product Line Conference - Volume B, Paris, France, September 9–13, 2019 (SPLC '19),* 8 pages.

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342416

https://doi.org/10.1145/3307630.3342416

1 INTRODUCTION

Software Product Lines Engineering (SPLE) takes advantage of the common features of a family of products and anticipates the expected degree of variation over the product's lifetime [5],[13]. This means that SPLE exploits the commonality found in the products of the same family by using reusable core assets and the variability of the family by constructing variable assets (typically developed in a phase called domain engineering [13]), which are then customized and assembled into specific products (typically developed in a phase called application engineering [13]). SPLE implies a big-upfront investment in domain engineering, but it turns on a beneficious ROI (Return of Investment) during the application engineering phase.

The complexity of current software systems makes non-functional requirements (NFR) at least as critical as functional ones. NFR are characteristics of the system that are provided as a whole, so they commonly crosscut the functionality of the system. NFR are classified from different points of views. Sommerville [17] classifies them as external, organizational and product requirements; whereas the standard ISO/IEC/IEEE 29148 [10] divides them into quality and stakeholder requirements. Depending on the standard, they are categorized in a different way, and those NFR that are highly related with the architecture and design are also called as quality attributes [17].There are standardized quality models to evaluate this quality attributes [9]. Standards and guidelines confirmed by standard committees can be used as a mechanism to measure in which degree a software product fulfils a NFR or quality attribute.

The functional variability in SPL has been extended using specific quality or NFR information. In this paper, we focus on the NFR standards and guidelines, specifically in the Accessibility Web Content Accessibility Guidelines (WCAG) 2.1, W3C Recommendation [22]. This work presents the specification of the accessibility features and information that the WCAG pre-establishes being product and family independent. Therefore, it can be used for any SPL related to web accessibility.

Since accessibility crosscut the functionality of products and their families, this work defines an Accessibility Variability Model, which is interconnected with a functional Variability Model [13]. In particular, it has been represented using Orthogonal Variability Models (OVM). In order to illustrate this contribution, it has been adopted in the domain engineering phase by constructing the SPL of OER family, and by deriving the Massive Open Course (MOOC) initiative of the Universidad Técnica Particular de Loja [19].

The rest of the paper is structured as follows: Section 2 provides the background about accessibility; Section 3 provides an overview

of related works; Section 4 shows the definition of accessibility OVM; Section 5 an experience study case in order to apply the accessibility OVM; and Section 6 presents conclusions and future work.

2 ACCESSIBILITY

Accessibility is a key non-functional property in any application and even more so in the Web. Tim Berners-Lee argues that 'The social value of the Web is that it enables human communication, commerce, and opportunities to share knowledge'.

One of the World Wide Web Consortium [21] primary goals is to make these advantages available to all people. According to ISO 9241-11:1998 [12], accessibility is the use of a product, service, framework or resource in an efficient, effective, and satisfying way by people with different abilities . To obtain a quantitative assessment of web accessibility, some methods, standards and international guides are available. The Accessibility Evaluation Methods (AEM) may differ in terms of effectiveness, efficiency and utility [2]. Regarding standards and norms, the most representative and globally referenced are those presented in the W3C Web Accessibility Initiative (WAI) [20], that brings together people from industry, disability organizations, governments, and research centers of the world to develop guidelines and resources to help make the web more accessible to people with disabilities, from speech to auditory, cognitive, neurological, physical and visual disabilities. The WAI initiative comprises standards, guidelines and techniques in different versions. This work is focused on Websites and web applications - Web Content Accessibility Guidelines (WCAG 2.1) [22]. WCAG 2.1 covers a wide range of recommendations to make web content more accessible, and these recommendations also make web content more useful for common users. These guidelines establish three levels to structure and guide the accessibility evaluation of web contents as follow:

- Principles: the foundations of web accessibility such as perceivable, operable, understandable and robust.
- Guidelines: twelve guidelines that refine the principles. These guidelines provide the basic objectives that authors must achieve in order to create more accessible content for users with different levels of disability.
- Success Criteria: for each of the twelve guidelines, the WCAG 2.1 provides verifiable compliance success criteria.

These guidelines are used to evaluate requirements and needs such as design specifications, purchasing, regulation or contractual agreements. To conform to WCAG 2.1, it is necessary to satisfy the success criteria guaranteeing that there is no content that violates them (see Table1). The evaluation of success criteria is performed in terms of three levels of conformance (see Table1), in such a way one of the three is met in full:

- Level A (lowest): The web page satisfies all the Level A Success Criteria, or conformance to an alternate version is provided.
- Level AA (medium): The web page satisfies all the Level A and Level AA Success Criteria, or a Level AA alternate version is provided.

Table 1: Web Content Accessibility Guidelines WCAG 2.1 (Adapted of [22])

PRINCIPLE / Accessibility guidelines WCAG 2.1	**WCAG 2.1 Accessibility criterion**	**Accessibility level**
PERCEPTIBLE: Information and user interface components must be presentable to users in ways they can perceive.		
1.1 Text alternatives: Provide text alternatives for any non-text content so that it can be changed into other forms people need, such as large print, braille, speech, symbols or simpler language.	1.1.1 Non–textual content: All non-text content that is presented to the user has a text alternative that serves the equivalent purpose, except for the situations listed below	A
1.2 Based on mean time: Provide, alternatives for time–based means of communication.	1.2.1 Audio–only and video–only (Prerecorded): For prerecorded audio-only and prerecorded video-only media, the following are true, except when the audio or video is a media alternative for text and is clearly labeled.	A

- Level AAA (highest): The web page satisfies all the Level A, Level AA and Level AAA Success Criteria, or a Level AAA alternate version is provided.

In addition to evaluate the accessibility, WAI also provides resources that support and help in its assessment. In previous work [14], authors presented a methodology for evaluating the accessibility and usability of OER sites. This methodology is based on the WCAG. This methodology reveals that these guidelines can be represented as an OVM model. In this work, we are going to use the WCAG 2.1 and before their OVM representation, we have made a synthesis of their principles, accessibility guidelines and success criteria definitions and their accessibility level in a Table (see an excerpt of this in Table1, see the complete specification in the supplementary material[1]).

3 RELATED WORKS

There are many approaches that define how to specify NFR in SPL, some of them are based on Feature Models (FM) and include NFR with an extended FM, as the work of Benavides et al. [1]. This work proposes to model SPL considering both, functional and non-functional features. FM are extended with attributes [7], and their relationships together with dependencies relationships between features [18]. So, functional requirements and NFR coexist in the

[1] https://bit.ly/2JR9PyN

same model. The works of Chavarriaga et al. [4], [3] propose operations to address the need of combining FMs. This work is based on the premise that in the manufacturing sector, it is required the use of different FMs for specifying the domain variability and the standard and regulations variability of manufactured products, such as transformers of power networks. On the other hand, there are other approaches based on UML models, the work of González-Huerta et. al [8] presents a MDD approach that allows the identification and representation of quality NFR for SPL. The approach makes use of a multimodel to express through a quality NFR metamodel. This work takes a step forward by defining a metamodel that allow the specification of any NFR, specifically focusing on quality. However, this work presents the quality requirements models as UML class models conformed to the metamodel instead of using variability models. The work of Nguyen et. Al. [11] consists of the integration of: a) the extension of Product Line UML-Based Software Engineering (PLUS) [11] to analyze and quantify NFR, and b) a reasoning engine. This integration constitutes a unified and systematic framework for analysis modeling of NFR in SPLs. Another approach is [23], where a systematic approach for modelling quality attributes in feature models and making quality-aware configurations is defined. This approach is implemented by a three-phase process: (1) Identify and represent the quality attributes in the feature models, (2) Measure the interdependencies between the features and the quality attributes, (3) Configure a product considering the quality.

There are other approaches based on Orthogonal Variability Models (OVM) for modelling NFR. Roos-Frantz et. al [15] present an approach for analyzing the quality of an SPL. This work represents the variability with an OVM that it is mapped with the quality model to associate the quality information and allow the constraints verification. In addition, authors have developed the tool FaMa-OVM to support the modelling of OVM [1], [16]. These related works vary in the model that they use for specifying variability, but they also have points in common. The works of González-Huerta [8] and Roos-Frantz [15] set out the need of managing quality as an orthogonal aspect. However, they do not provide guidelines about the non-functional features that should be considered to deal with a specific NFR. González-Huerta [8] proposes a NFR metamodel and Roos-Frantz [15] proposes quality attributes associated to the OVM, where features or attributes that are non-related with the NFR can be defined. So, it is necessary to have control not only of the syntax, but also the semantics by guiding the user in which features and attributes are related to a specific NFR or not. Therefore, we propose non-functional specific OVM to guarantee the alignment between the non-functional features and the NFR standards and guidelines, in particular the accessibility OVM. As a result, this work provides a guidance by establishing which are the accessibility features that have to be considered. This accessibility OVM is complementary to Functional OVM, as the work of Chavarriaga et al. [4], [3] propose for specifying the variability of manufactured products with multiple FM.

4 ACCESIBILITY OVM

In this paper, we present the accessibility VM using an OVM representation. OVM provide a cross-sectional view of the variability across all product line artefacts [13] by interrelating the variability with base models such as requirement models, design models, component models, and test models. The traceability between OVM and the different types of base models is established through dependencies [15], which allow us to know the impact and implementation of the variable features into software products.

Accessibility OVM has been specified based on the standard WCAG 2.1 (see section 2.1). The principles and guides are modelled as variation points (VP) or variants; whereas the success criteria are modelled as variants (V). Figure 1 represents the main variability points of the Accessibility OVM and their variants, and the variants of the Perceivable variability. It is not feasible to illustrate all the VP and variants due to space reasons (see complete model in the supplementary material[2]). In the Accessibility OVM, every variability is optional because it is an option to make a website accessible, and also vary to what extent accessibility is implemented (see Figure 1 and the supplementary material). The accessibility OVM is reusable for any family related to web contents, since it is based on the WCAG standard. This accessibility OVM defines those criteria and principles of the standard to guarantee the standard compliance and its reusability.

The OER SPL has a large functional OVM, where the courses, teachers, and content management conform the core of the SPL and vary the different ways of providing these common functionalities, in addition to extra functionalities. Figure 3 illustrates an excerpt of the SPL, just to illustrate the relationship between the functional and non-functional variants from the functional and accesibility OVM, respectively. For example, the OER content is related the Perceivable principle, therefore some variants of the VP Content Management are related with the variants of the Perceivable VP (see Figure 3).

In addition, the quality model to measure the accessibility of products has been defined. The quality model is presented in Table 2. Each quality attribute is defined by an identifier, a name, a domain, a principle/guide, an accessibility level and the formula to calculate the value. The quality attributes are related with their corresponding variability points or variants of the Accessibility OVM (see Figure 2 and it's supplementary material [3]). As shown in Table 2, the accepted values for attributes associated to variants may be from 0 to 3. The standard WCAG 2.1 categorizes its success criteria between A, AA, or AAA. The formula to obtain the values for variants are: IF (A) = 1, IF(AA)=2, IF(AAA)=3, see the examples in Quality Information of Figure 2 1.2.1, 1.2.5, 1.2.7, respectively. Some variants are constrained by another VP, for example, the variant V.1.3 Adaptable (see Figure 1). In this case the formula depends on the VP formula. Therefore, in order to support the constrained variants, the domain is always established between 0 and the maximum value that the established by the accessibility level (see Table 2 and Figure 2), that is 0 to 3. On the other hand, the VP formula is the average of its variants. Once the Accessibility OVM is configured the value of the formulas are calculated.

The defined quality model results will provide us information about the accessibility degree of a specific product and for each of the accessibility principle. If the result is between 0 and 1, the product will be labelled with an A accessibility, if the result is between

[2] https://bit.ly/2YFt9Dn
[3] https://bit.ly/2HDyl3U

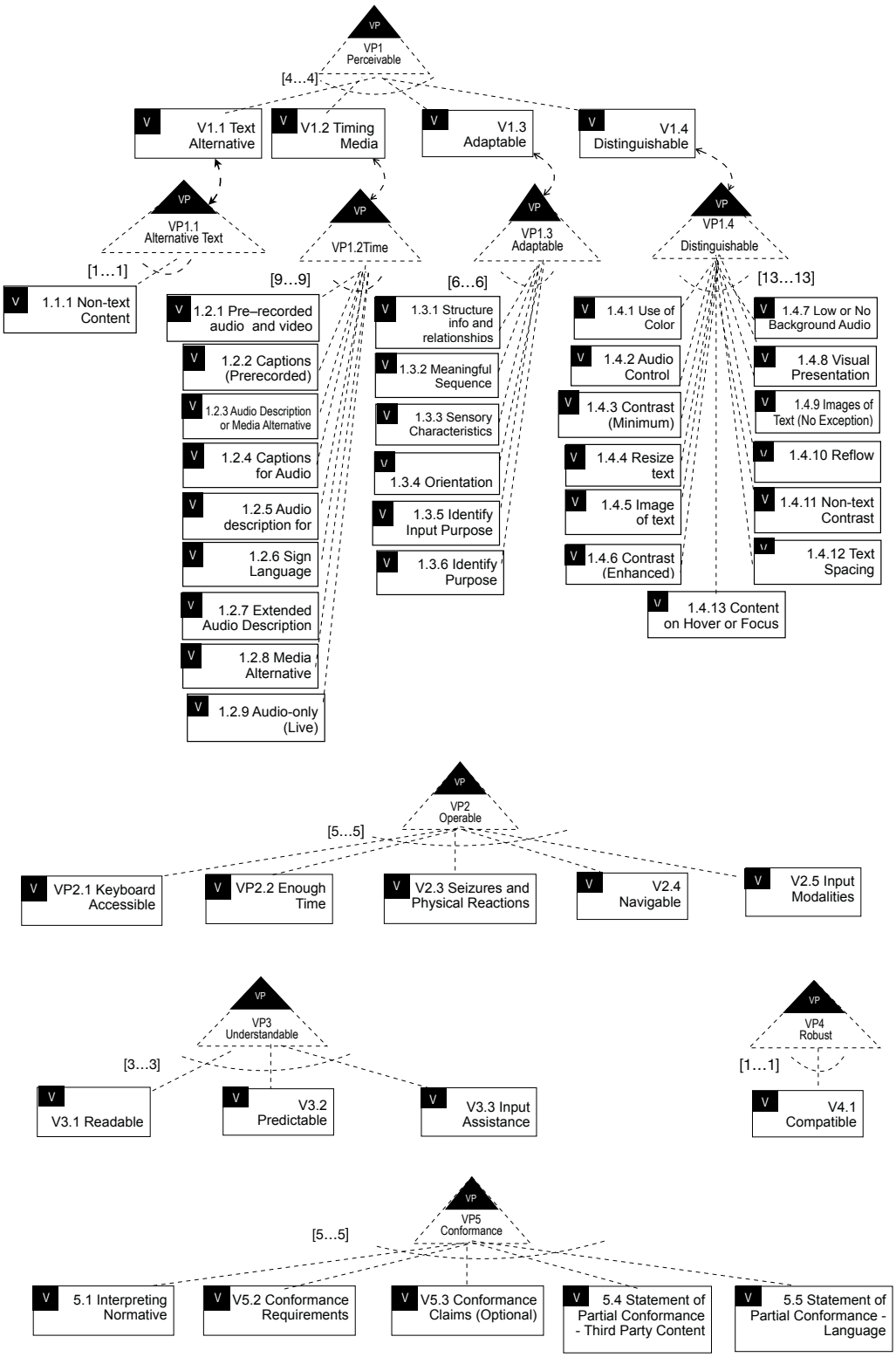


Figure 1: Accessibility OVM (Excerpt)

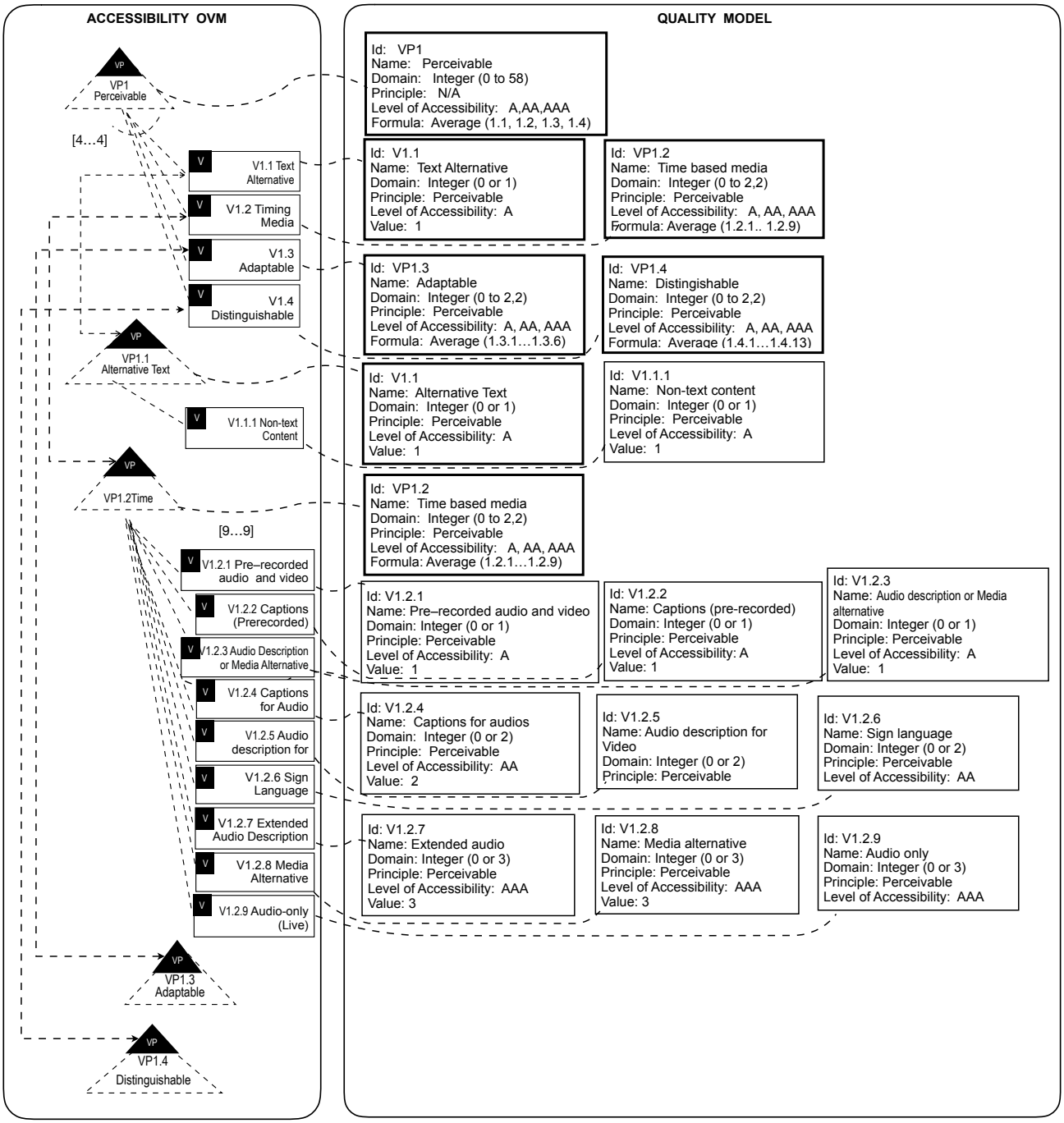


Figure 2: Accessibility OVM and quality model relationships

1 and 1,38, the product will be labelled with an AA accessibility, and finally, if the result is between 1,39 a 1,96 the product will be labelled with an AAA accessibility.

It is important to emphasize that this accessibility OVM together the defined quality model can be used by web designers to evaluate existing web sites or knowing the accessibility degree during requirements elicitation. This is a valuable information, since they

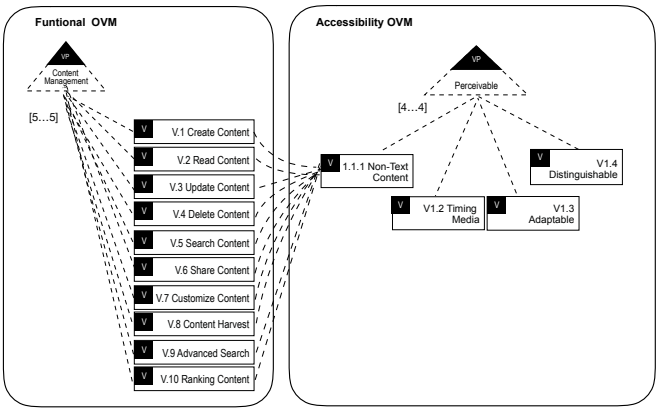


Figure 3: Functional OVM and Accessibility OVM relationships

Table 2: Quality information Model

Name	Label	Description
Identifier	Id	Criteria/principle/guide number in the standard WCAG 2.1
Name	Name	Criteria/principle/guide Name in the standard WCAG 2.1
Domain	Domain	Range of acceptable values for measure the criteria/principle/guide
Principle	Principle	Accessibility principle or guide that the variant is related to.
Accesibility Level	Accesibility Level	Accessibility level for each criteria: A, AA, AAA
Value / Formula	Value / Formula	Variants: Percentages assigned by level of Accessibility A = 0.65%, AA = 1.31%, AAA = 1.96% VP: The value is obtained by adding the percentages of the associated variants, so that the total sum of all the variants of the LMO is 100%

are on time to correct requirements if they realize that the product does not have the desirable accessibility degree. So, they can select more features or more valuable features in advance for being implementing by following guidelines to improve accessibility [14].

5 ACCESIBILITY OVM INTO PRACTICE: THE UTPL MOOC

Evidence of the viability of the accessibility OVM can be obtained by putting the model into practice in a real-life setting. Therefore, we have conducted a case study.

5.1 Case Study Description

As a first experience, this work has been applied the accessibility OVM of a web site to evaluate its accessibility degree. Instead of developing a web site from scratch, we have derived the configuration of an existing MOOC [24] from the SPL. Specifically, the derived product is the MOOC of the Universidad Técnica Particular de Loja (UTPL) [7]. The UTPL MOOC has 12 available courses and it is composed of 100 web pages that have many functionalities such as content management, user management, searches, user interface facilities etc.

5.2 Research Objective

As the completeness of the accessibility features is already guaranteed by the fact that they are defined by standard/standardized guidelines, we know that the accessibility OVM considers the required features for modelling accessibility in web content applications. Therefore, in this initial pilot case study, we focused on the accessibility measurement to assist the engineer during the derivation of a product from the SPL, and thus, obtaining a product that maximizes the accessibility. In this pilot case study, the main two questions to be answered through the case study analysis can be formulated as follows:

- **RQ1.** Are the accessibility OVM and its quality information model able to provide a measure of the accessibility of the system from the derivation of a product?
- **RQ2.** Is able this mechanism to recommend which features should be selected during the derivation to improve the accessibility of a previous derived product?

5.3 Reporting

RQ1. The first step was to configure the product of the MOOC UTPL from the OER SPL by deriving during the application engineering the functional and accessibility configuration. In particular, the derivation was performed from the main page and some MOOC random pages. The result of this derivation of the Accessibility Model is presented in Figure 4.

From the selected variants of the Accessibility OVM illustrated in Figure 4, the quality attribute information is calculated during the configuration phase. Table 3 illustrates the accessibility evaluation results after the application engineering for this product. The values for each variability point are obtained applying the formulate associated to it which is the average of the success criteria value for example the VP1.3 is the result of average of their variants V1.3.1, V1.3.2, V1.3.3, V1.3.4, V1.3.5, V1.3.6 in the study case none of these variants is present in the case of studies for that the VP1.3 value is 0. A total average of 0,18 means that it has an A accessibility level of average, but not all the variants of A are implemented. This is an evidence that the proposed model allows providing a measure and a label of accessibility to a specific product and to reveal that the site should be improved.

RQ2. As a second exercise and based on previous works and our experience in the area [6], we have used the model to configure a product by selecting the minimum desirable accessibility features that a product of this family should have to be accessible. To that end, we have configured the product by selecting the variants presented in Figure 5, and we have obtained the evaluation presented in Table 4. This product may be used as a reference in the area, to reach this minimum value of accessibility level, i.e. an average AA. As a result, it may help engineers recommending including the features that are desired features to improve accessibility. For example, in our case it is recommend to improve the Perceivable

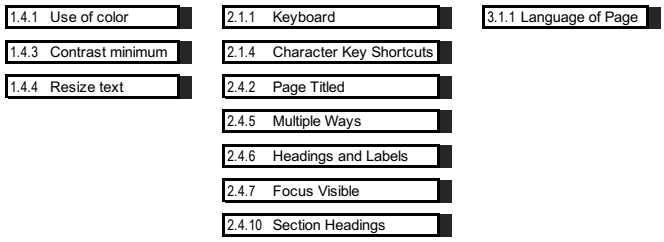


Figure 4: Current UTPL MOOC configuration

Table 3: Accessibility Evaluation Results of the UTPL MOOC

Variability Point		Average
VP1 Perceivable		1.00
VP1.1	Text Alternative	0.00
VP1.2	Time based media	0.00
VP1.3	Adaptable	0.00
VP1.4	Distinguishable	0.38
VP2 Operable		0.26
VP2.1	Keyboard Accessible	0.50
VP2.2	Enough Time	0.00
VP2.3	Seizures and Physical Reactions	0.00
VP2.4	Navigable	0.80
VP2.5	Input Modalities	0.00
VP3 Understandable		0.05
VP3.1	Readable	0.16
VP3.2	Predictable	0.00
VP3.3	Input Assistance	0.00
VP4 Robust		0.33
VP4.1	Compatible	0.33
Total		**0,18**

principle by including 22 new features such as: Non-text content, Pre–recorded audio and video, Pre–recorded audio and video, etc.; 7 new features to address the Operable principle; 10 new features in the Understandable principle; and finally, to address the Robust principle that it is missing in the current version of the product.

6 CONCLUSION AND FUTURE WORK

This paper presents how to specify the variability of NFR standardized by standard committees using specific OVM, specifically the accessibility OVM of the web content accessibility guides (WCAG) 2.1 W3C recommendation. This contribution allows one to guarantee following the WCAG and having a measure of their compliance, since the quality model associated to them provide this information. In addition, this paper demonstrates its adoption by presenting the Accessibility OVM and its application in the Open Educational Resources (OER) SPL by deriving and measuring a specific product, the UTPL MOOC. Finally, it is important to emphasize that this work also provides a product model with the minimum desirable properties to consider a product accessible enough. This product model in terms of accessibility may be a guide to reach or exceed for those products accessibility-aware. This work is the beginning

Table 4: Accessibility desirable feature level for MOOCs sites

Variability Point		Average
VP1 Perceivable		1.47
VP1.1	Text Alternative	1.00
VP1.2	Time based media	1.77
VP1.3	Adaptable	1.66
VP1.4	Distinguishable	1.46
VP2 Operable		1.04
VP2.1	Keyboard Accessible	1.50
VP2.2	Enough Time	0.33
VP2.3	Seizures and Physical Reactions	1.00
VP2.4	Navigable	1.20
VP2.5	Input Modalities	1.17
VP3 Understandable		0.74
VP3.1	Readable	0.50
VP3.2	Predictable	0.40
VP3.3	Input Assistance	1.33
VP4 Robust		1.33
VP4.1	Compatible	1.33
Total		**1,16**

of a wide variety of future work, from automatizing the defined process, to extend the case study and to specify more Non-Functional Specific OVM for other standards.

ACKNOWLEDGEMENTS

This work has been partially funded by by Secretaria Nacional de Ciencia y Tecnología (SENESCYT) Ecuador, the Project CROWDSAVING (TIN2016-79726-C2-1-R), the EU FEDER program, the MINECO project OPHELIA (RTI2018-101204-B-C22); the TASOVA network (MCIU-AEI TIN2017-90644-REDT); and the Junta de Andalucia METAMORFOSIS project.

REFERENCES

[1] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. *Seminal Contributions to Information Systems Engineering* 01 (2005), 361–373. https://doi.org/10.1007/978-3-642-36926-1{_}29
[2] Giorgio Brajnik. 2008. A comparative test of web accessibility evaluation methods. (2008), 113. https://doi.org/10.1145/1414471.1414494
[3] Jaime Chavarriaga, Rubby Casallas, and Viviane Jonckers. 2017. Implementing Operations to Combine Feature Models: The Conditional Intersection Case. *Proceedings - 2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design, VACE 2017* (2017), 41–47. https://doi.org/10.1109/VACE.2017.2
[4] Jaime Chavarriaga, Carlos Rangel, Carlos Noguera, Rubby Casallas, and Viviane Jonckers. 2015. Using multiple feature models to specify configuration options for electrical transformers. (2015), 216–224. https://doi.org/10.1145/2791060.2791091
[5] Paul Clements and Linda Northrop. 2002. *Software Product Lines: Practices and Patterns.* Addison-Wesley.
[6] Krzysztof Czarnecki. 2002. Generative Programming : Methods , Techniques ,. (2002), 351–352.
[7] Krysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley Professional. 864 pages.
[8] Javier González-Huerta, Emilio Insfran, Silvia Abrahão, and John D. McGregor. 2012. Non-functional requirements in model-driven software product line engineering. December (2012), 1–6. https://doi.org/10.1145/2420942.2420948
[9] ISO 25010. [n. d.]. *International Standard, Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.* Technical Report.
[10] ISO/IEC/IEEE 29148. 2011. *International Standard, Systems and software engineering — Life cycle processes — Requirements engineering.* Technical Report.
[11] Quyen L Nguyen. 2009. Non-functional requirements analysis modeling for software product lines BT - 2009 ICSE Workshop on Modeling in Software Engineering, MiSE 2009, May 16, 2009 - May 24, 2009. (2009), 56–61. https://doi.org/10.1109/MISE.2009.5069898

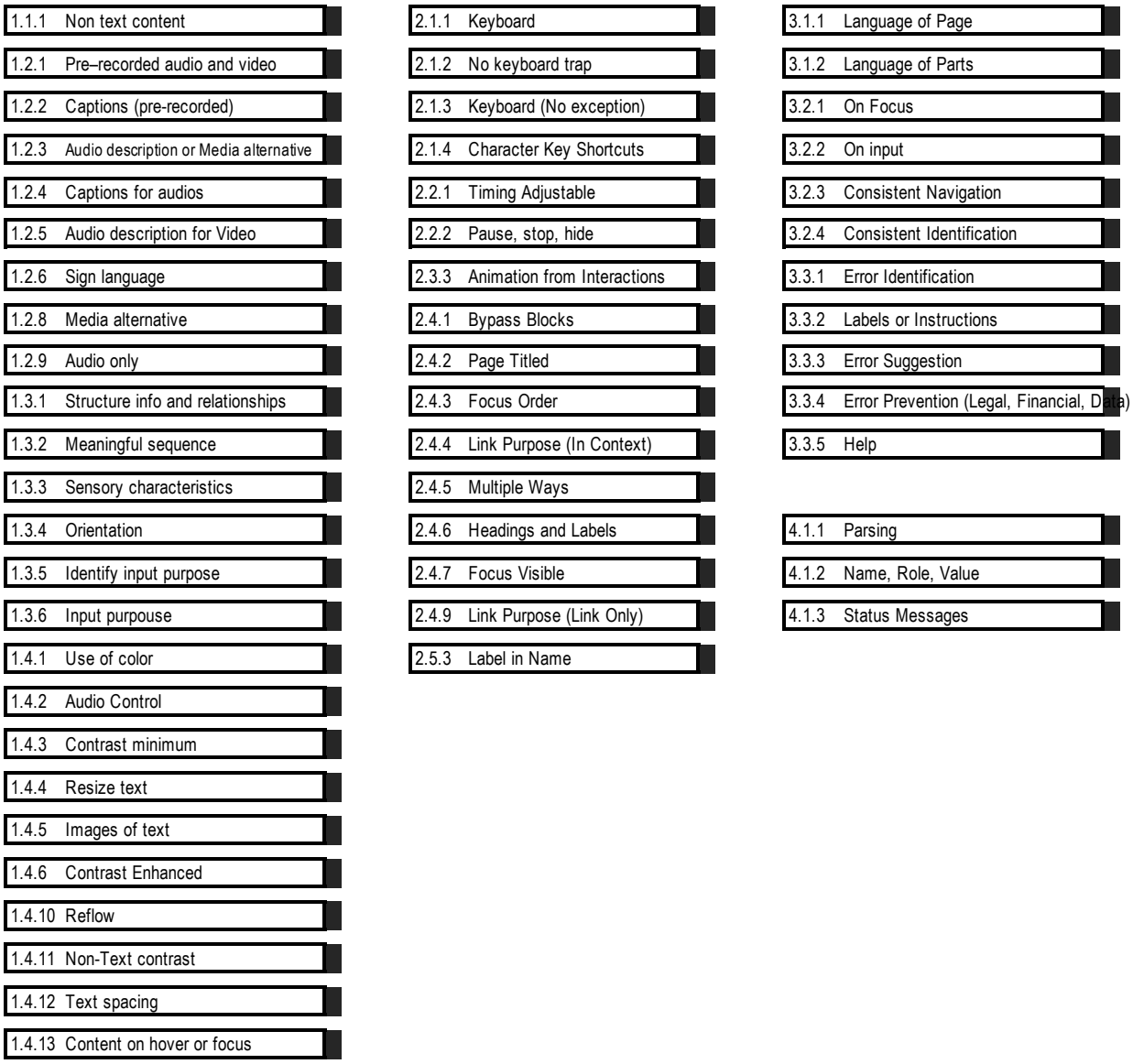


Figure 5: The minimum desirable Accessibility in a MOOC configuration

[12] International Standard Organization. 1998. *ISO 9241-11 Guidance on usability.* Technical Report.

[13] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering Foundations, Principles and Techniques.* Springer.

[14] G. Rodríguez, J. Pérez, S. Cueva, and R. Torres. 2017. A framework for improving web accessibility and usability of Open Course Ware sites. *Computers and Education* 109 (2017). https://doi.org/10.1016/j.compedu.2017.02.013

[15] Fabricia Roos-Frantz, David Benavides, Antonio Ruiz-Cortés, André Heuer, and Kim Lauenroth. 2011. Quality-aware analysis in product line engineering with the orthogonal variability model. *Software Quality Journal* 20, 3-4 (2011), 519–565. https://doi.org/10.1007/s11219-011-9156-5

[16] Fabricia Roos-Frantz, José A Galindo, David Benavides, and Antonio Ruiz Cortés. 2012. FaMa-OVM: a tool for the automated analysis of OVMs. *16th International Software Product Line Conference, {SPLC} '12, Salvador, Brazil - September 2-7, 2012, Volume 2* (2012), 250–254. https://doi.org/10.1145/2364412.2364456

[17] Ian Sommerville. 2016. *Software Engineering.* Pearson Education.

[18] Detlef Streitferdt, Matthias Riebisch, and Ilka Philippow. 2003. Details of Formalized Relations in Feature Models Using OCL Detlef Streitferdt. In *Proceedings of 10th IEEE International Conference on Engineering of Computer–Based Systems (ECBS 2003).* 45–54.

[19] UTPL. 2019. UTPL MOOC. (2019). https://mooc.utpl.edu.ec/

[20] W3C. [n. d.]. Web Accessible Initiative. ([n. d.]). https://www.w3.org/WAI/

[21] W3C. [n. d.]. World Wide Web Consortium. ([n. d.]).

[22] W3C. 2018. Web Content Accessibility Guidelines (WCAG) 2.1. (2018). https://www.w3.org/TR/WCAG21/

[23] 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

Reusability in Artificial Neural Networks: An Empirical Study

Javad Ghofrani
Faculty of Informatics/Mathematics
HTW University of Applied Sciences
Dresden, Germany
javad.ghofrani@gmail.com

Ehsan Kozegar
Faculty of Engineering (Eastern Guilan)
University of Guilan
Guilan, Iran
kozegar@guilan.ac.ir

Arezoo Bozorgmehr
Uni Klinikum Bonn
Bonn, Germany
arezoo.bozorgmehr16@gmail.com

Mohammad Divband Soorati
University of Lübeck
Lübeck, Germany
divband@iti.uni-luebeck.de

ABSTRACT

Machine learning, especially deep learning has aroused interests of researchers and practitioners for the last few years in development of intelligent systems such as speech, natural language, and image processing. Software solutions based on machine learning techniques attract more attention as alternatives to conventional software systems. In this paper, we investigate how reusability techniques are applied in implementation of artificial neural networks (ANNs). We conducted an empirical study with an online survey among experts with experience in developing solutions with ANNs. We analyze the feedback of more than 100 experts to our survey. The results show existing challenges and some of the applied solutions in an intersection between reusability and ANNs.

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; • **Software and its engineering** → **Reusability**; *Software product lines*; • **General and reference** → *Empirical studies.*

KEYWORDS

Systematic reuse, artificial neural networks, reusability, survey, empirical study

ACM Reference Format:
Javad Ghofrani, Ehsan Kozegar, Arezoo Bozorgmehr, and Mohammad Divband Soorati. 2019. Reusability in Artificial Neural Networks: An Empirical Study. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3307630.3342419

1 INTRODUCTION

Artificial neural networks (ANNs), especially deep neural networks (DNNs), have shown fast raising success in last few years. Contribution of DNNs in various fields such as speech processing, natural language processing, and computer vision was significant. Deep learning has been a powerful technique in the edge of technologies including smart production systems [22], smart cities [9], and self-driving cars. Recently, the world's biggest companies (e.g., Google, Amazon, and Microsoft) started to invest in DNN based solutions. Tensorflow [1] and CNTK [36] are two of many frameworks offered by companies to facilitate and standardize the development of ANN-based software systems.

Big data analytics and Convolutional Neural Networks (CNNs) drew the attention of researchers and practitioners to ANN techniques. Instead of writing complicated algorithms, software systems are developed based on training the neural networks on large datasets [25]. Despite outstanding performance and satisfying functionality in special contexts (e.g., image classification or prediction of future events in industrial systems), ANNs is not able to solve every challenge in the domain of software engineering (SE) [31].

Development of software systems with ANNs is not exploited enough by the community of (SE) [6]. Most of the concepts in SE are applied in the development of software products based on algorithms, codes, and logical test cases. One of the most important concepts in every stage of software development is reusing software artifacts [33]. Libraries of codes [30], models [13, 19], and meta-models are some of the reusable components that software engineers build. Reusability is a well-established method in the development of software systems in various fields and ecosystems [14, 18]. Systematic reuse as the main concept of software product lines (SPLs) leads to higher quality, less costs, and faster time-to-market [5]. In such software intensive systems, general-purpose building blocks (i.e., core assets) are created to be reused in development of a family of similar software systems. Nevertheless, utilization of reusability concepts is neglected in most sub-domains of developing ANNs. Efforts were made to develop libraries such as Keras [11] and PyTorch [32] to foster the development of ANN-based solutions. However, many aspects of SE are not considered in such solutions. For example, there are several concerns including stability, integrity issues, and conflicts between the dependencies of these libraries. These frequently occurring issues show a lack of maturity in using SE methods for developing ANN tools and frameworks. We assume that the reasons behind the existing gap are: (i) low quality codes written and maintained during the development of ANN-based solutions compared to conventional software systems; (ii) dependency of ANN-based solutions to their application

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342419

domains; and (iii) lack of awareness among ANN experts about the SE community and vice versa.

In this paper, we aim for bridging the gap between the communities of SE and ANN. We investigate the state of practice and existing challenges of applying reusability methods in development of ANN-based systems. We study how practitioners and researchers reuse the tools, solutions, and assets and how often reusability was considered while developing ANNs. We formulated our research questions as following:

- **RQ1: How are neural networks already reused?** The motivation behind this question is to estimate the state of reuse among experts of ANN. This helps us to find shortages, weaknesses, and strengths of reuse in this field.
- **RQ2: What types of neural networks are commonly reused and why?** Some of the neural networks are reused more often than the others. Understanding the reasons and factors behind it helps us to realize the features that leads to this preference.
- **RQ3: What are the main challenges related to the reuse of ANNs?** This question specifies the issues with reusing ANNs. A solution to these challenges can pave the way for a significant improvement of reusability in ANN-based projects.

In order to answer these questions, we conducted an empirical study using an online survey following the guideline proposed by Fink [12]. We collected and analyzed the opinion of 112 practitioner who are involved in the development of ANNs. Our contribution is to point out several issues that researchers can resolve to improve the quality of applying reusability techniques in the development of software systems based on ANNs.

In Section 2 we review existing approaches on supporting reuse in development of ANN-based solutions. Section 3 presents the details of our survey. Research questions were answered based on the collected feedback in Section 4. Discussions and conclusions are presented in Section 5 and 6.

2 RELATED WORK

Several studies proposed the utilization of ANNs to support SPLE activities such as Requirements Engineering [28] and management of variability and configurations [7]. Ghamizi [15] also introduced an end-to-end framework for systematic reuse of DNN architectures to create a product line of ANN-based software solutions. To the best of our knowledge, there are no notable empirical studies in the context of combining the reusability concepts and developing ANN-based solutions. However, efforts have been made to foster the development of ANN-based systems [4, 29, 34]. In this section, we address the existing approaches on supporting reusability in the development of ANN-based solutions.

Transfer learning is a common technique in reusing models among DNNs [16, 39] where a model that is trained on a data set will be trained further on another data set as well. This method saves time, costs, and resources for training a DNN. However, this type of reuse is only feasible for models that were trained on a more general data-set [40]. Transfer learning will be utilized in two common situations: lack of enough data or limited resources required to train a model of DNN. ImageNet [26] is one of the most used examples of transfer learning in the domain of image classification. This model is trained to classify 1000 classes of objects in the images and can be extended for other purposes in the domain of image classification. Similar models are also provided by Oxford [1] [37], Google [2] and Microsoft [3] [20]. Transfer learning is not limited to the models of image processing. Reuse of Models for language processing are also supported through approaches such as word2vec Model[4] from Google and GloVe Model [5] provided by Stanford.

A slew of machine learning frameworks are created to handle the complexity in development of ANNs based systems. A handful and easy to use set of APIs are available that support a fast development of ANNs. For instance, PyTorch [6] library provides flexibility in the development of DNNs in Python by facilitating the building of computational graphs. Another tool is TensorFlow [2] which is a framework to handle the complexity of development tasks on ANN-based solutions. In comparison to the other frameworks, it supports wider range of programming languages and platforms. CNTK [36] also is an open-source toolkit provided by Microsoft for handling the tasks related to development of deep-learning based software. Amazon adapted Apache MXNet [10] on its web services due to the scalability of the tool and the number of languages that it supports in the development of deep learning tasks.

3 SURVEY

3.1 Preparation

We had a brainstorming session to prepare the questions about reusability. The questions were:

(1) What is your understanding about reusability?
(2) How does your team apply reuse? on which components? and in which situations? any details?
(3) Which components were reused in your development processes? and why?
(4) What are the positive effects that you try to achieve with reuse? how important are those effects?
(5) Do you or your team apply any tools to support reuse and reusability? Why did you choose this/these tool(s)?

We asked seven experts in SE and three in ANN domains to answer these questions. We collected their answers and categorized them using a few tags. Based on these answers, we extracted questions with ready-to-select options to make the answering process as easy as possible. In the next step, we performed pilot tests with five participants who took part in our survey. The feedbacks and recommendations were then considered to finalize the survey. The survey is available online in PDF format [7]. Furthermore, the anonymized and filtered version of results for our survey is publicly available [8].

[1] http://www.robots.ox.ac.uk/~vgg/research/very_deep/
[2] Google Inception Model: https://github.com/tensorflow/models/tree/master/research/inception
[3] https://github.com/KaimingHe/deep-residual-networks
[4] https://code.google.com/archive/p/word2vec/
[5] https://nlp.stanford.edu/projects/glove/
[6] https://pytorch.org/
[7] https://doi.org/10.6084/m9.figshare.8152664.v3
[8] https://doi.org/10.6084/m9.figshare.8230178.v1

We used LimeSurvey software provided by the education portal of Saxony in Germany[9] to publish our survey and collect responses. The survey was active for seven weeks (from April 8th, 2019 to May 15th, 2019)[10]. We investigated scientific databases, including IEEE Explore, Springer, and ACM to create a list of papers related to ANNs. We sent out more than 1000 emails to the authors of these papers. We also distributed calls for participation within groups and forums of machine learning in ResearchGate, Linkedin, Twitter, Xing and Google.

3.2 Structure

The first page of the survey is filled with a short introduction similar to the content of the email which explains the goal of the survey. The questions with multiple choice answers have a free text field which enables the participants to write their answers the desired answer is not provided already as an option.

The survey consists of five steps: First step (Section A) includes a question to filter the non-relevant participants from the survey:

- A1. Have you ever used or implemented any artificial neural networks in any form?

In the second step (Section B) of the survey, four questions were proposed to collect information about "Most significant experience with ANNs". These questions are listed here:

- B1. When was your most significant project with ANNs?
- B2. How big is the team that you work with on this project (including yourself)?
- B3. Did you use any systematic development process(es) in this project?
- B4. In what way did you work at most with ANNs?

Third section of the survey (Section C) extracts details from participants about their opinions and experiences of reuse in ANNs which consists of seven questions:

- C1. How do you define reusability?
- C2. How often do you reuse the following parts related to artificial neural networks? (code, data-set, and parameters)
- C3. What is the most significant source related to artificial neural networks that you reuse for your projects?
- C4. In your projects, which of the following conditions trigger reuse?
  - First phase of each project
  - When I don't have enough knowledge about the domain
  - I know the domain and I am sure that similar work exists
  - If stakeholder(s) asks for reusing
- C5. How often do you reuse the following kinds of artificial neural networks? (Shallow Artificial Neural Networks (such as MLP, RBM, SOM, RBF, etc.), Deep Convolutional Neural Networks (CNNs), Deep Recurrent Neural Networks (RNNs), Deep Generative Adversarial Networks (GANs), Auto-encoders, Deep Belief Networks, Capsule Neural Networks, Other types of Deep Neural Networks)
- C6. How much do you agree with the following statements about reusing ANNs?
  - Searching for useful reusable modules is complicated/time consuming
  - Making modules reusable is complicated/time consuming
  - Modules are poorly documented, making it difficult to use
  - It is difficult to find out whether existing modules fulfills the task
- C7. How important is reuse in your process predominantly?

In step 4 (Section D), we collected the demographic data about participants and their working domain. Furthermore, we asked them about their background and experience in the development of ANNs. Final step (Section E) was to gather personal feedback about the survey to improve our future work. This step also contains an additional question about contact details to send the results of the survey and get back to the participants for more detailed questions if needed.

4 RESULTS

We received a total number of 229 responses including 114 complete and 115 incomplete surveys. We applied two filters on responses of the survey to increase the quality of our analysis. First filter removed the 115 incomplete surveys since they did not offer a significant information. The second filter discarded two responses from the remaining 114 surveys in which the participants did not select *Yes* as an answer to the filter question *A1*. Applying the previous filters led to 112 surveys in total which are considered to answer the proposed research question in Section 1.

4.1 Demographics

According to the results of the profiling part, we have collected the following answers.

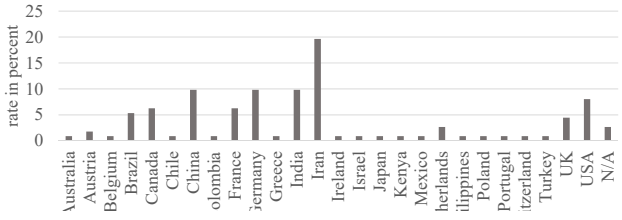


Figure 1: Rate of participants from different countries

As Figure 1 shows, most participants are from Iran (22 participants, 19.64%). China, Germany, India have 11 participants (9.83%) . USA has the third highest participation in our list with 9 participants (8.04%). France and Canada with 7 participants (each 6.25%) are fourth. Brazil with 6 participants (5.36%), UK with 5 participants (4.46%), Netherlands with 3 participants (2.68%) sit at the lower part of our list. We had also 17 (15.13%) participants from other countries: Japan, Pakistan, Australia, Switzerland, Kenya, Poland, Austria, Austria, Mexico, Greece, Chile, Turkey, Belgium, Ireland, Israel, Colombia, Philippines.

Most participants (64 participants, 57.14%) are computer scientists. 37 participants (33.04%) chose "engineering" as their field of

[9] https://bildungsportal.sachsen.de/portal/
[10] https://bildungsportal.sachsen.de/umfragen/limesurvey/index.php/782756?lang=en

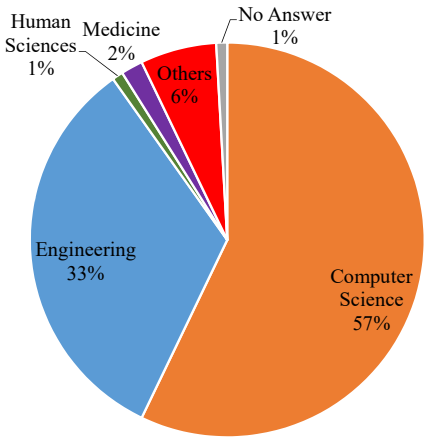


Figure 2: Participants towards their field of study

study. We have 11 participants from other fields such as bio-medical engineering, physics, mathematics and medicine (see Figure 2).

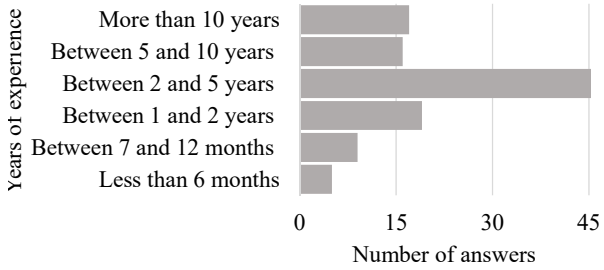


Figure 3: Years of experience with ANN

Almost half of the Participants (46 participants, 41.07%) have between 2 and 5 years of experience/background with ANNs (See Figure 3). Based on the result in Figure 4, 109 participants used ANNs in the context of research, 37 in teaching, 42 in industrial context, and 24 for hobby, There are few answers about working with ANNs in art (2 participants), 15 in private, and 2 in other contexts. Selecting more than one answer to this question was allowed.

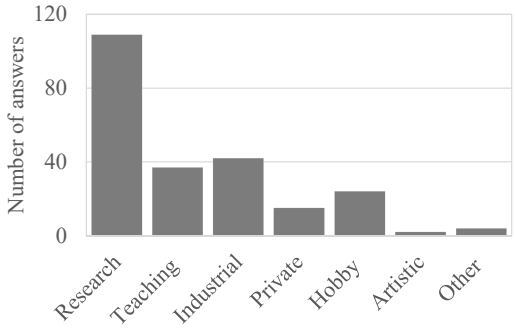


Figure 4: Field of experience with ANN

Question B3 asked participants about development processes that are used in most significant project that they had with ANN. As Figure 5 illustrates, ad-hoc process are used most by the teams that worked with ANNs (63 answers), while some of them used systematic development processes (33 answers).

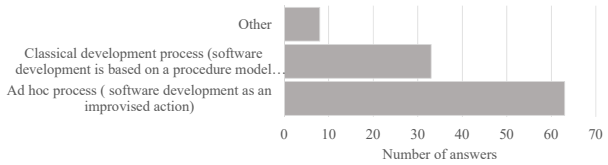


Figure 5: Utilized Development Process in the most significant ANN-based Project

Demographic data shows that many participants have between 2 and 5 years of experience. Most participants come from the fields of computer science and engineering, and they are active in research, teaching, and industry. Furthermore, applying ad-hoc processes for managing the projects is more common among participants in comparison to classical development processes.

4.2 RQ1: How are neural networks already reused?

According to the answers to C1 (Shown in Figure 6), 79 of 112 participants define reusability as "Modify an existing artificial neural network to adopt it to a specific context", while 53 of 112 participants define reusability as "Create a general neural network with the aim of using in various contexts (e.g. convolutional neural networks that have been trained on data-sets like ImageNet, without any changes)". Note that the participants had also the option to select none or both definitions.

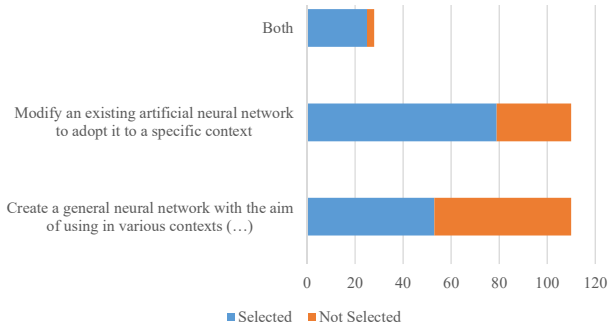


Figure 6: Selected Answers for the question "How do you define Reuse"

Reuse can occur in three ways through ANNs: data-set, network structures (i.e., code), and parameters (i.e., weights). In the first way, the data-set from third party will be reused to train an ANN that the user have developed on his own. Based on the answers to C2, reuse occurs "Often" in "data-set" level among participants. This is the most common abstraction level of reuse in ANNs. Second stage of reuse in ANNs is reuse of specific architectures (such as Alexnet[26],

VGG[37], GoogleNet[38], ResNet[20], and DensNet[23]) that can be utilized as network structures (i.e., code) . In this level, an ANN is trained using an arbitrary training set. The weights of the network (parameters) should be tuned to achieve a high degree of accuracy. Participants "Often" reused the structure. Third level of reuse is increased to the values stored in the structure of the network (i.e., parameters). In this approach, a pre-trained network including its layers and weights (i.e., parameters) will be used completely or partially (i.e., transfer learning). Based on the results of question C2, participants "Sometimes" reuse parameters of ANN in their projects.

Figure 7 illustrates that all three parts related to ANNs (code, data-set, and parameters) are somehow reused. Code and data-set are reused more often compared to parameters. Answers to the question C2 confirm that all participants perform reuse in at least one of the three abstraction levels of ANNs.

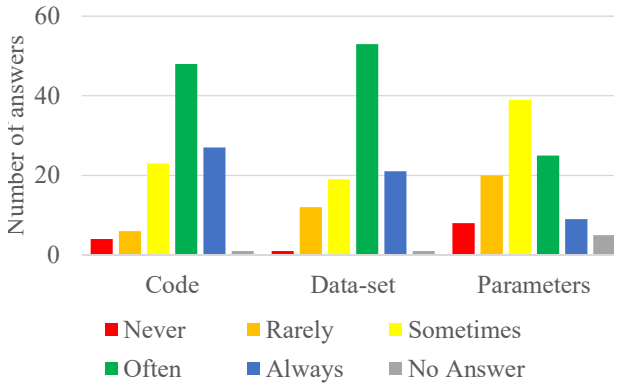


Figure 7: How often different parts related to ANN are reused

It should be noted that most of the participants (79%) use Public repositories, 34% use Private own collections. 16% received it from colleges, and 11% through Other developers via private communications.

Even though the benefits of reuse has been well established in software development, understanding the reasons behind the utilization the reuse in ANN-based projects provide a comprehensive explanation about situations where reuse can be triggered. Therefore, question C4 then asks the participants which conditions trigger reuse in their projects. Figure 8 shows that 76 participants selected the option "I know the domain and I am sure that similar work exists" as the trigger for reuse. 42 participants chose "First phase of each project" and 32 participants mentioned "When I don't have enough knowledge about the domain". 8 participants mentioned that the demand from the stakeholders was the motivation for reuse by selecting "If stakeholder(s) asks for reusing". It should be noted that participants were free to choose more than one of the listed answers.

Question C7 asks how valuable reuse is for participants. It is "Not at all important" for 4.14% of our participants. 23.21% of them chose "Slightly important", 36.61% of participants found it "Important". Finally, it is "Fairly Important" for 18.75% as well as "Very Important" for 11.61% of participants. In total, 66,96% of participants find reuse "important", "fairly important", or "very important".

In summary, reuse is important in the development of ANNs and it happens in all main levels of ANNs' structure. However, the network structures and data-sets will be reused more than the parameters. Furthermore, there are more tendency to modify an ANN and reuse it instead of providing reusable networks.

4.3 RQ2: What types of neural networks are commonly reused and why?

Results of initial interviews and the investigation in literature guided us to eight common types of ANNs that experts reuse in their works. The common types of the NNs and their frequency of reuse is shown in Figure 9. Among proposed ANNs, Deep Belief Networks (DBNs) [21] and Capsule NNs [35] have the most answers on getting "never" reused. However, Deep Convolutional Neural Networks (CNNs) [27] are reused ANNs among participants the most.

We calculated weighted mean values for the Likert scale [3] for each ANN category. Based on the resulting mean values, CNNs get "sometimes" reused. While Deep Recurrent Neural Networks (RNNs), Shallow artificial NNs (such as multilayer perceptron, restricted Boltzmann machine, self-organizing map, radial basis function) [24], auto-encoders [8], and Deep Generative Adversarial Networks (GANs) [17] are reused "rarely".

4.4 RQ3: What are the main challenges related to the reuse of ANNs?

Question C6 measures the degree of agreement of participants with following statements:

- (SQ1) Searching for useful reusable modules is complicated/-time consuming
- (SQ2) Making modules reusable is complicated/time consuming
- (SQ3) Modules are poorly documented, making it difficult to use
- (SQ4) It is difficult to find out whether existing modules fulfills the task

Figure 10 illustrates the overview of the answers collected for this question. A small group of participants is more than "Agree" with this statement. Calculating the average values for given scales, reveals that all four statements are "Neutral" among participants. There are considerable "Disagree" and "Neutral" answers to the statements. Most answers agree with both statements in SQ2 and SQ4. These statements consider the difficulties in creating reusable components and finding whether the existing modules fulfills the task. Despite the agreement of 33% of participants with SQ1, 27% of participants disagree that searching for reusable modules is complicated or time consuming. Level of agreement and disagreement is decreased for SQ3. It shows that 16,96% "Disagree" with the problems around documentations of reusable modules, while 33,04% are "Neutral" and 27,68% "Agree".

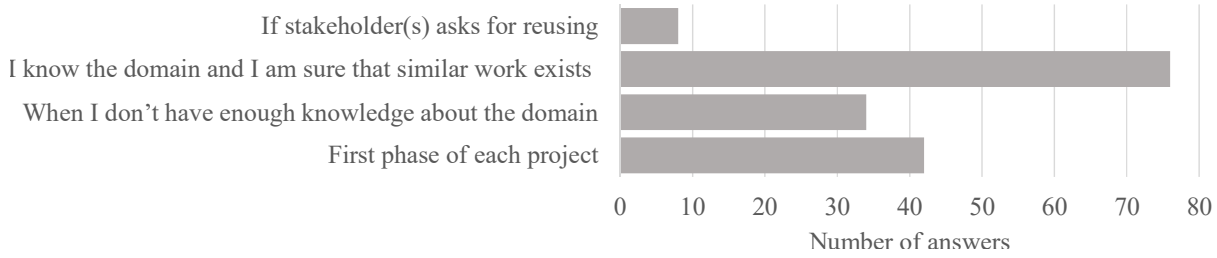


Figure 8: Number of answers to C4 about factors which trigger reuse

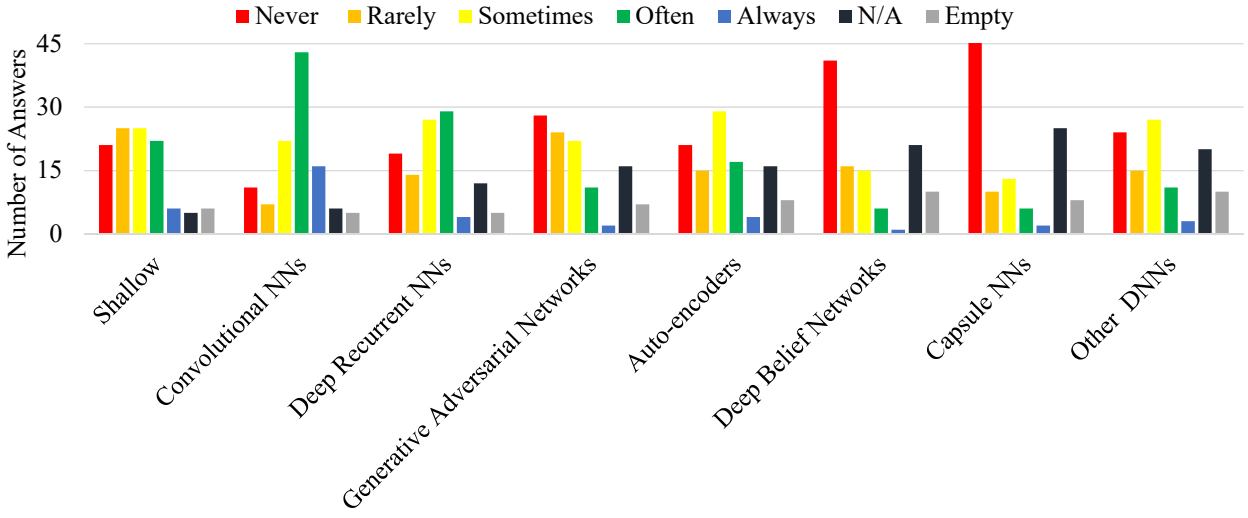


Figure 9: Frequency of reuse for common ANNs

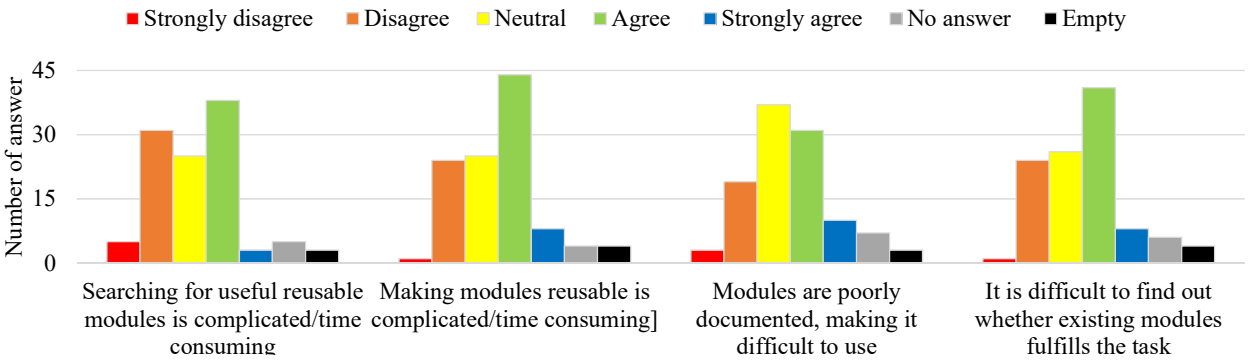


Figure 10: Agreement level of the participants (in percent) towards four statements on challenges in applying reuse to ANNs

5 DISCUSSION

The period of data gathering with our survey was arguably enough as we tried to motivate experts as much as possible. We carefully designed the survey questions to be short but unequivocal so that the participants would not be bored or disappointed.

The number of participants to our survey is 112 which is not a huge set of experts. One reason was that we filtered out many and tried to reach only those that were qualified enough through published scientific works and expert forums. We discarded uncompleted submissions from our results to improve the quality of the survey.

A significant proportion of experts in our survey are from a research community which was predicted as many industrial partners do not publish the contact details of their employees. We tried to reach as many as we could via LinkedIn and online forums. We do not consider the lack of participant from industrial sector to have negative impact on our study since the main objective of our research was to bridge the gap between SE and ANN communities.

Several questions could improve the study that we left out to keep the survey short. Asking these questions could improve the quality of answers. However, there is a risk of reducing the number of participants since the survey takes longer to fill. Furthermore, we collected the email addresses of the interested participants for further questions. We will contact them for detailed questions and create an extension of our study for future work. It may be useful for further studies to know (i) with which kind of deep neural network do they often work, (ii) in their opinion which challenges exists, and (iii) suggestions to improve the reusability of ANNs.

Answering all questions in the field of ANNs and reusability is beyond the scope of our research. Our main objective is put a spotlight on the state of reusability in development of ANN-based projects. Considering the results of our survey can motivate other researchers to create further research questions.

We have considered central tendency bias in addressing the multiple-choice questions by providing examples about proposed answers. However, some questions do not have enough examples or details that could be a bias for choosing an answer. The problem in this case is that there is no proper examples. We avoid using odd-numbered scales with a middle value which could confuse the participants.

Regarding to C5, we categorized the common ANNs based on their conceptual differences in existing implementations of ANNs. That is why some of the well-known ANNs such as Long Short Term Memories (LSTMs) are not included since it is a subset of deep recurrent NNs. Furthermore, we should mention that some of the results were unexpected for this question. For example, implementing a Capsule Net or GANs from scratch is very complicated. One reason that significant portion of answers to the frequency of reuse of such ANNs is "Never" or "Rarely" is the lack of reusability support for these types of ANNs in official distribution of common libraries and tools such as Keras. Reuse in deep convolutional NNs (CNNs) is dominant compared to other architectures listed in C5. This may be because CNNs are the first group of deep ANNs [27] that addressed the issue of computational complexity by utilizing GPUs [26]. Wide range of application domains for these types of ANNs resulted in various tools and frameworks that facilitated their applications.

Mentioned challenges in C6 (i.e., RQ3) are based on the comments collected from preliminary interviews and the pilot test. We were aware of the need for further research to provide a systematic overview on existing challenges, but none of participants left comments about potential challenges which may be ignored in the survey. As shown in Figure 10, the number of participants who "Agree" or "Disagree" in SQ2, SQ3, and SQ4 is considerably different. For example, regarding "It is difficult to find out whether existing modules fulfill the task" 49 participants selected "Agree" or "Strongly agree". 25 participants selected (strongly) disagreed. Based on answers to C1 (see Figure 6), we assume that their disagreement is because of their different perceptions or insufficient understanding of reusability. Both options in C1 are valid definitions of reusability concept, yet a large number of participants did not select both (see Figure 6). Different opinions show that the ANNs experts not only have fundamentally different views, which could be a signal of the lack of knowledge in understanding important concepts—reusability—which exists in the SE domain.

Answers to C7 reveals that reuse is important among experts of ANN. However, the results illustrated in Figure 7 show the low rate of reuse for different types of ANNs (except CNNs) among participants.

Although we asked about systematic development process(es) in B3, further investigations about the knowledge of participants in software development methods is required. Regarding close and fast-growing collaboration between SE and ML, a systematic approach for assessment, validation, and standardization of the level of knowledge among experts from both fields seem necessary.

Besides the important factors mentioned in RQ1, biases and hyperparameters (e.g., number of epochs and learning rates) also play a crucial role in ANNs. We assume that reusing them is not common as none of our interviewees and participants mentioned them in their answers. Therefore, we did not include reusability of biases and hyperparameters in our survey.

We performed an empirical study on reusability applied to the domain of ANNs. However, introducing and investigating all aspects of SPLs (e.g., variability models, configuration, feature interactions) in ANNs and machine learning is not the focus of this paper.

6 CONCLUSIONS

In this paper, we investigated the state of reuse among experts of ANNs using empirical results collected from an online survey in 2019 with 112 participants. Three main research questions were answered: (i) how neural networks will be reused? (ii) what kind of neural networks are commonly reused? and (iii) what are the common challenges related to the reuse of ANNs? Although reuse is applied in the development of ANN-based solutions, the level of reuse and the maturity of knowledge among experts of ANN are arguably low. We found that codes and data-sets related to the development of ANNs are reused more often in comparison to parameters. Furthermore, the public repositories (e.g., GitHub) are the most attended references for reuse. A large number of participants find reuse essential for their projects although they have different opinions about reuse and reusability. Participants mainly apply reuse in their domains of expertise Otherwise, they prefer to develop new solutions from scratch. It seems to be obvious that deep CNNs have more reused compared to other common architectures of ANNs.

Creating reusable modules is one of the challenges that should be considered. Finding out whether existing modules fulfill the requirements of the task is another challenge.

Modules are often poorly documented that makes the utilization of reuse more complicated. The result of our empirical study points out an opportunity for two communities of SE and ANNs for knowledge transfer. We base our future work on bridging the gap especially by establishing the domain of reusability for ANNs.

ACKNOWLEDGMENTS

This project is co-financed with funds on the basis of the budget adopted by the deputies of the Saxon state parliament

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[3] I Elaine Allen and Christopher A Seaman. 2007. Likert scales and data analyses. *Quality progress* 40, 7 (2007), 64–65.

[4] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 39–48.

[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-oriented software product lines*. Springer.

[6] Anders Arpteg, Björn Brinne, Luka Crnkovic-Friis, and Jan Bosch. 2018. Software engineering challenges of deep learning. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 50–59.

[7] Davide Bacciu, Stefania Gnesi, and Laura Semini. 2015. Using a machine learning approach to implement and evaluate product line features. *arXiv preprint arXiv:1508.03906* (2015).

[8] Pierre Baldi. 2012. Autoencoders, unsupervised learning, and deep architectures. In *Proceedings of ICML workshop on unsupervised and transfer learning*. 37–49.

[9] Michael Batty, Kay W Axhausen, Fosca Giannotti, Alexei Pozdnoukhov, Armando Bazzani, Monica Wachowicz, Georgios Ouzounis, and Yuval Portugali. 2012. Smart cities of the future. *The European Physical Journal Special Topics* 214, 1 (2012), 481–518.

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[11] François Chollet et al. 2015. Keras. https://keras.io.

[12] Arlene Fink. 2015. *How to conduct surveys: A step-by-step guide*. Sage Publications.

[13] William Frakes and Carol Terry. 1996. Software reuse: metrics and models. *ACM Computing Surveys (CSUR)* 28, 2 (1996), 415–435.

[14] William B Frakes and Kyo Kang. 2005. Software reuse research: Status and future. *IEEE transactions on Software Engineering* 31, 7 (2005), 529–536.

[15] Salah Ghamizi, Maxime Cordy, Mike Papadakis, and Yves Le Traon. 2019. Automated Search for Configurations of Deep Neural Network Architectures. *arXiv preprint arXiv:1904.04612* (2019).

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[17] 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.

[18] Martin L Griss, I Jacobson, and P Jonsson. 1998. Software reuse: Architecture, process and organization for business success.. In *TOOLS (26)*. 465.

[19] Haitham Hamza and Mohamed E Fayad. 2002. Model-based software reuse using stable analysis patterns. ECOOP.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385* (2015).

[21] Geoffrey E Hinton. 2009. Deep belief networks. *Scholarpedia* 4, 5 (2009), 5947.

[22] Hartmut Hirsch-Kreinsen. 2014. Smart production systems. A new type of industrial process innovation. In *DRUID Society Conference*. 16–18.

[23] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[24] Anil K Jain, Jianchang Mao, and KM Mohiuddin. 1996. Artificial neural networks: A tutorial. *Computer* 3 (1996), 31–44.

[25] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. 2007. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering* 160 (2007), 3–24.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[27] Yann LeCun, LD Jackel, Leon Bottou, A Brunot, Corinna Cortes, JS Denker, Harris Drucker, I Guyon, UA Muller, Eduard Sackinger, et al. 1995. Comparison of learning algorithms for handwritten digit recognition. In *International conference on artificial neural networks*, Vol. 60. Perth, Australia, 53–60.

[28] Yang Li, Sandro Schulze, and Gunter Saake. 2018. Extracting features from requirements: Achieving accuracy and automation with neural networks. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 477–481.

[29] Jie Lu, Vahid Behbood, Peng Hao, Hua Zuo, Shan Xue, and Guangquan Zhang. 2015. Transfer learning using computational intelligence: a survey. *Knowledge-Based Systems* 80 (2015), 14–23.

[30] Ali Mili, Rym Mili, and Roland T Mittermeir. 1998. A survey of software reuse libraries. *Annals of Software Engineering* 5, 1 (1998), 349–414.

[31] Helena Holmström Olsson and Ivica Crnkovic. [n. d.]. A Taxonomy of Software Engineering Challenges for Machine Learning Systems: An Empirical Investigation. In *Agile Processes in Software Engineering and Extreme Programming: 20th International Conference, XP 2019, Montréal, QC, Canada, May 21–25, 2019, Proceedings*. Springer, 227.

[32] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch. In *NIPS Autodiff Workshop*.

[33] Ruben Prieto-Diaz and Peter Freeman. 1987. Classifying software for reusability. *IEEE software* 4, 1 (1987), 6.

[34] Joseph Reisinger, Kenneth O Stanley, and Risto Miikkulainen. 2004. Evolving reusable neural modules. In *Genetic and Evolutionary Computation Conference*. Springer, 69–81.

[35] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. 2017. Dynamic routing between capsules. In *Advances in neural information processing systems*. 3856–3866.

[36] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2135–2135.

[37] K. Simonyan and A. Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014).

[38] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[39] 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.

[40] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks?. In *Advances in neural information processing systems*. 3320–3328.

Nine Years of Courses on Software Product Lines at Universidad de los Andes, Colombia

Jaime Chavarriaga
Rubby Casallas
Universidad de los Andes
Systems and Computing Department
Bogotá, Colombia
ja.chavarriaga908@uniandes.edu.co
rcasalla@uniandes.edu.co

Carlos Parra
Pontificia Universidad Javeriana
Systems Engineering Department
Bogotá, Colombia
ca.parraa@javeriana.edu.co

Martha Cecilia Henao-Mejía
Carlos Ricardo Calle-Archila
Universidad de los Andes
ConectaTe, Faculty of Education
Bogota, Colombia
mc.henaom@uniandes.edu.co
cr.calle@uniandes.edu.co

ABSTRACT

Software Product Lines has been taught in Universidad de los Andes, Colombia, since 2011. The content, activities and evaluation in these courses have changed during this period of time. For instance, while topics such as the processes to engineer product lines, feature models to specify domain variability, and design patterns to implement the variability are common to all these courses, other topics such as the product line maturity levels, some techniques to implement variability and recent automation practices for testing, continuous integration and delivery have varied with the time. In addition, topics and activities, such as the course project that has been present in all the courses, had also been modified. This paper (1) describes the evolution of our courses on Software Product Lines, presenting commonalities and variabilities in their topics, activities and evaluation techniques and (2) discusses some lessons learned during its recent design as a Blended Learning course.

CCS CONCEPTS

• **Software and its engineering** → *Software product lines.*

KEYWORDS

Software Product Lines, Variability, Teaching

ACM Reference Format:
Jaime Chavarriaga, Rubby Casallas, Carlos Parra, Martha Cecilia Henao-Mejía, and Carlos Ricardo Calle-Archila. 2019. Nine Years of Courses on Software Product Lines at Universidad de los Andes, Colombia. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3307630.3342415

1 INTRODUCTION

For the last nine years, we have taught *Software Product Lines* as a main subject in courses offered to undergraduate and postgraduate students. These courses have evolved, not only in their content and topics, but also in the activities to perform and the implementation techniques to study.

On the one hand, the evolution of these courses is a consequence of the changes that we have witnessed in the corresponding theories, approaches and tools. For instance, while our initial course at 2011 had lectures and exercises related to *Orthogonal Variability Models (OVMs)* and used *SPLOT* as a variability modeling tool, most recent courses are more focused on Feature Models and use *FeatureIDE* for modelling and development tutorials and in-class projects.

On the other hand, the courses have evolved in the pedagogical approaches we use. The last edition has been implemented as a Blended Learning course, i.e., mixing traditional face-to-face education with online learning, to make it more accessible and flexible, give student control on the pace of the activities, develop soft skills such as time management and critical thinking, and to promote the engagement of teachers and students.

This paper describes (1) the evolution of our courses identifying commonalities and variabilities among them and (2) our recent experience designing it as a blended learning course. In addition, we discuss some lessons learned and provide some hints for people interested on designing blended courses on Software Product Lines.

Rest of this paper is organized as follow: Section 2 describes the evolution of our courses. Section 3 presents our experience designing the course as a blended learning one and Section 4 describes the resulting design. Finally, Section 5 discusses some lessons learned and Section 6 concludes the paper.

2 EVOLUTION OF OUR SOFTWARE PRODUCT LINES COURSES

We have taught ten courses during the last nine years: Three courses, from 2011 to 2013, in the Master of Systems Engineering and seven courses, from 2014 to 2019, in the Master of Software Engineering. The descriptions of all these courses are available in the *Course Catalog*[1] of the Universidad de los Andes.

To understand their evolution, we have grouped the courses in three periods. The following are brief descriptions of each period:

Software Product Lines (2011-2013). Initial courses were lead by Rubby Casallas. The "Software Product Lines" course (ISIS-4715)[2] was offered from 2011 to 2013. These courses were part of the Master of Systems Engineering and was also available to

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342415

[1] https://catalogo.uniandes.edu.co/
[2] https://profesores.virtual.uniandes.edu.co/~isis4715/dokuwiki/

undergraduate students as an elective course. It was a sixty-six week course with a weekly 3-hours session.

The course content was heavily influenced by the Pohl et al.'s book [10] on Engineering Product Lines, and the SEI's framework [6]. It included topics like: (1) domain engineering and product engineering, (2) *feature models* to specify external variability, (3) *orthogonal variability models (OVMs)* to specify internal variability, and (4) the *Family Evaluation Framework (FEF)* to determine the maturity level of an organization regarding SPL practices.

Course activities included a review of the taxonomy for variability implementation techniques proposed by Svahnberg et al.[11], several hand-on-labs on conditional compilation, aspects and design patterns to implement variability, and a course project where groups of students implemented a SPL using these techniques. In addition, in 2011 and 2012, students visited diverse companies using some the FEF-based instruments to estimate their SPL maturity level.

Software Factories and Product Lines (2014-2019). Later, the courses lead by Rubby Casallas and Carlos Parra, incorporated patterns and techniques to implement product lines and ecosystems considering not only variability but also extensibility and automation. The course also changed its name to "Software Factories and Product Lines" (MISO-4204)[3] to reflect these changes. The course remained as a sixty-six week course with a weekly 3-hours session.

The course content was influenced, not only by the Pohl's book [10] on Software Product Lines, but also by the Greenfield's "Software Factories" approach [8] to develop product lines and the Czarnecki's [7] and Benavides et al.'s work [3] on feature models. On the one hand, several topics related to designing extensible software, automate product derivation and develop software configurators were included. On the other hand, the course had more emphasis on modeling and analyzing feature models and the study of OVM models was reduced gradually.

The course included activities to cover the new topics. For instance, new activities were included to implement product lines using different techniques, to extract information from annotated source code, and to perform manually some analyses on feature models. In addition, the course project was modified to allow students to use any language for development and require the implementation of variability with, at least, three different techniques. Some of the techniques used at this time included REST Services, Aspect Oriented Programming, Design Patterns, Code Generation, binary replacement, condition-on-constant and condition-on-variable among others. In addition, instead of using SPLOT[4] and VarMod[5] to create Feature and OVM models, the course started to use FeatureIDE[6] to model and implement product lines in its activities.

Software Factories and Product Lines (2019-today). Recently, the "Software Factories and Product Lines" (MISO-4204) course was redesigned by Jaime Chavarriaga and Rubby Casallas to (1) include topics regarding modern practices for testing, continuous integration and delivery and (2) be taught as a blended course. Now, the course combines in-classroom and online activities. Students perform weekly activities with an estimated effort of 12 hours. Almost all the weeks, these activities are completely online. Each tree weeks, the students attend a 3-hour in-classroom session and perform online activities estimated in 8 hours. The first course in this modality was taught the first semester of 2019 and it will be offered again starting in August 2019.

The course content, although it maintains the influence of the SPL engineering processes proposed by Pohl et al., it is highly influenced by the Capilla's [5] and Apel's [2] books on Variability and Product Lines, and the Meinicke et al.'s book [9] on development using FeatureIDE. Furthermore, the course includes new topics such as testing, continuous integration and delivery of software product lines, and some case-studies considering experiences in real-life companies.

In contrast to the previous courses, a large number of tutorials and videos were built to allow students to learn and understand technical concepts and practices on their own. The students have access to online materials that explain the topics and describe the requirements for the diverse projects in the course.

3 DESIGN AS A BLENDED LEARNING COURSE

We spent more than one year designing "Software Factories and Product Lines" as a blended learning courses. In a joint project with *Centro de Innovación en Tecnología y Educación (Conecta-TE)*[7], we defined a working group with two lecturers of the course, a pedagogue and a technical expert on designing courses using Learning Management Systems. In addition, the resulting course was evaluated by two experts on educational design.

Following the method proposed by Conecta-TE, the design of blended learning was conceived as a two-dimensional process [1]: On the one hand, a *pedagogical and curricular dimension* where the designers decide on the teaching methods and learning styles to apply in the courses. And, on the other hand, a *technological dimension* where the designers decide on the tools and materials, such as videos, tutorials and simulations, to design and implement as activities in the course.

For each dimension, the design process follows the *ADDIE model*, i.e., it comprises five stages [4]: (A) Analysis, (D) Design, (D) Development, (I) implementation and (E) Evaluation.

Analysis For the "Software Factories and Product Lines" course, analysis started at 2018-1 collecting information from the SPL courses taught in the University since 2011, and from courses and materials of other universities. At beginning of 2018-2, we established the scope, learning objectives and pedagogical and didactic considerations for the course.

Design During 2018-2, we defined the structure of the course, explored diverse learning paths for each topic, and designed the diverse activities in the course. In weekly meetings, we (two lecturers, a pedagogues and an engineer) worked together analyzing the topics and determining activities that can be used to teach them. We specified the results in multiple documents and spreadsheets. In addition, we sketched videos, tutorials and home-works in the process,

[3] https://profesores.virtual.uniandes.edu.co/~miso4204/dokuwiki/
[4] http://www.splot-research.org/SPLOT
[5] http://www.sse.uni-due.de/varmod
[6] https://featureide.github.io/
[7] https://conectate.uniandes.edu.co/

Development We started to implement the activities into a Moodle platform at the end of 2018-2.

Implementation We started our first blended course at 2019-1 with fourteen (14) students. We keep doing weekly meetings to track and analyse the progress and improve the course.

Evaluation A comprehensive evaluation: two focal group discussions and a survey were performed during the course in 2019-1. Evaluation results are being used to improve the course design and activities in the next term.

4 CURRENT COURSE DESIGN

Nowadays, the "Software Factories and Product Lines" is a sixteen (16) weeks blended learning course. It has activities in a classroom every three weeks. In the middle, students perform activities such as tutorials, record videos and develop individual and group projects. All these activities are uploaded and tracked in online platforms such as a Moodle, Padlet and Github.

During the course, the students have permanent support of a *course tutor* that also evaluates some of the activities. Classroom activities are lead by the *course lecturer*. Instead of a typical lecture, classroom activities comprise discussions, presentations and group activities.

Course goals At the end of this course, students will be able to (1) implement business strategies based on software factories and product lines, (2) determine, model and analyze the variability in a set of software products, (3) design, implement and test a product line, and (4) analyze and discuss problems implementing these approaches in real contexts.

Course structure The course has classroom activities each three weeks. The course starts or ends a unit on each of these sessions. Online activities are focused on learning concepts and developing technical skills related to software factories and software product lines. These activities typically start with lectures and tutorials, follow with individual projects and end with a group project and a discussion. Classroom activities are focused on discussing results, analyzing criteria for making decisions in real companies and reflecting on the required personal skills and company processes.

Course content The course is organized in five sections:

(1) *An introduction to Software Factories and Product Lines*, where a background on product line concepts, engineering processes and multiple examples are discussed. Among other activities, the students do tutorials on FeatureIDE, build examples from the Spl2go, complete a software product line, record some video demonstrations and discusses differences to other types of software projects.

(2) *Variability Modeling and Analysis*, where software requirements and variability are modeled using diverse types of specifications, use cases and feature models. Activities include modeling diverse problems, analyzing consistency of multiple models, counting products in a feature model by hand and using the FeatureIDE API to perform some automated analyses.

(3) *Variability Implementation*, where multiple techniques for implementing variability are discussed. Activities include tutorials, individual projects and recording of video demonstrations. This section ends with multiple discussions on benefits of each technique and criteria that can be used to select one in a real-life project.

(4) *Course Project*, where students must create an initial software product line based on a case study. The students define a scope, design and develop the product line (domain engineering) and implement and test each product (application engineering). Further discussions are focused on problems applying the concepts and techniques during the project.

(5) *Integration to other Business Strategies*, where additional activities are focused on integrating product lines to business and technical strategies such as automated testing and continuous integration and delivery (CI/CD).

Teachers the course is taught by a *course lecturer*, currently a postdoc researcher of the University, and a *course tutor*, a teaching assistant with a 12 hours/week dedication[8].

5 LESSONS LEARNED

The course taught at the first semester of 2019 had a comprehensive evaluation using focal groups with students, many surveys and multiple reviews with experts. The following are some lessons learned in the process.

Regarding the blended learning design:

Blended learning is more than adding online activities All the course activities must be carefully designed to achieve the learning objectives, considering the strengths and weaknesses of each type of activity.

Classroom activities are different Instead of typical lectures, classroom activities such as presentations, analyses and discussions are focused on developing skills such as communication abilities, time management and critical thinking.

Students have more control on the pace of the course In contrast to traditional courses where the lecturer has a complete control, in blended learning each student work on his/her own rhythm. This can be confusing for lecturers used to keep all the students doing the same activity at the same time.

Personalized attention is key for engagement Students may feel that they are alone if they do not note that teachers are there for them. Teachers must participate and check students' works permanently to engage the students.

Blended learning design is an exhaustive process Each learning path must be carefully planned to provide coherence and consistency among the online and the classroom activities.

Multi-disciplinary design projects are beneficial During the course design, participation of pedagogues and technology experts help to consider different approaches to teach the course. They promote discussions aimed to determine the best types of activities for each topic.

Comprehensive evaluations are helpful Evaluations such as focal group give feedback about the strengths and weaknesses of both the learning paths and the activities therein. They

[8] In Universidad de los Andes, blended-courses require a course tutor for each thirty students.

gave us many areas to improve and help us to prioritize a "wish-list" for the following editions of the course.

Regarding the course design:

Students must understand the course design Considering that some courses are different to the others, it is very important that each student understands clearly the planned schedule and activities. Part of the classroom time and course videos must be focused on explaining the learning objectives and the activities.

Rubrics help students to plan their work Giving students the rubrics in advance, help them to understand what is the scope of the work and what elements have more relevance. Students may plan their work using this information. Some activities were performed to review the project grades, discuss how the work may be better organized and develop skills for planning projects considering their priorities and restrictions.

Students perform more individual work than the observed in face-to-face courses. Considering that the course proposes weekly activities with an estimated effort 12 hours, including tutorials, individual projects and group discussions; the general perception of the teachers is that the students do more work in blended learning than in traditional courses.

Teaching modeling requires more resources For modeling variability and requirements, we created a lot of resources: tutorials, videos, home-works, quizzes and evaluations. We designed a learning path where the activities increase its complexity and the students had opportunities to discuss their doubts and propose improvements to solutions presented by others.

Activities may help to discussing design criteria Learning paths where students do tutorials, create individual projects, compare techniques and participate in group discussions contribute to develop critical thinking and design criteria. We are using multiple sequences like to teach techniques for implementing variability and for designing software product lines.

Course projects require more attention Considering that students may advance (or not) in their group projects without requesting support of the teachers, it is necessary to define activities such as web meetings and review of deliverables to monitor the progress. We think that is useful to define groups, roles and responsibilities early in the project and define activities for tracking the progress considering these roles. Checklists and rubrics are very useful too.

6 CONCLUSIONS

We have presented the evolution of the courses taught at Universidad de los Andes, Colombia, regarding Software Product Lines. In this paper, we described the content, activities and evaluation of the ten courses taught since 2011. We noted that, while topics such as the processes to engineer product lines, feature models to specify domain variability, and design patterns to implement the variability are common to all these courses, other topics such as the product line maturity levels, some techniques to implement variability and recent automation practices for testing, continuous integration and delivery have varied with the time.

This paper also reports some lessons learned designing "Software Factories and Product Lines" as a blended learning course. This process required more than one year and involved two lecturers, a pedagogue, a technical expert and two evaluation experts. It required an exhaustive process to analyze the course subjects, design the learning paths and create the educational resources. The resulting course gave students more activities to learn the methods and technologies, discuss what decision make for specific scenarios and build more complex development projects. Comparing the results with previous editions of the course, the lecturers considers that the students made more individual work and performed more analytical work than previous editions of the course.

Nowadays, we are interested on improving our courses in future editions. We are considering to include more discussions on criteria and strategies to implement software product lines in existing companies. We are interested on discussing recent trends where companies implement variability in mobile applications and microservices using technoligies such as *Feature Toogles*, a.k.a. *Feature Flags* and centralized configuration databases. In addition, further work is being planned to design new and improve the existing resources and activities.

REFERENCES

[1] María Fernanda Aldana-Vargas and Luz Adriana Osorio. 2019. Pedagogical Guidelines for the Design of Blended Learning Environments. *Revista Internacional de Tecnologías en la Educación* 6, 1 (2019), 23–37.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.* Springer, Switzerland.

[3] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. In *Advanced Information Systems Engineering (CAISE 2005).* Springer Berlin Heidelberg, Porto, Portugal, 491–503.

[4] Robert K. Branson, Gail T. Rayner, J. Lamarr Cox, John P. Furman, F. J. King, and Wallace H. Hannum. 1975. *Interservice Procedures for Instructional Systems Development. Executive Summary and Model.* Technical Report. U.S. Army Training Board. https://apps.dtic.mil/docs/citations/ADA019486

[5] Rafael Capilla, Jan Bosch, and Kyo-Chul Kang. 2013. *Systems and Software Variability Management: Concepts, Tools and Experiences.* Springer, Switzerland.

[6] Paul C. Clements and Linda M. Northrop. 2002. *Software Product Lines: Practices and Patterns.* Addison-Wesley, Boston, MA, USA.

[7] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

[8] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. 2004. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* John Wiley & Sons, New York, NY, USA.

[9] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE.* Springer, Switzerland.

[10] Klaus Pohl, Gunter Bockle, and Frank van der Linden. 2005. *Software Product Line Engineering : Foundations, Principles, and Techniques.* Springer, Switzerland.

[11] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. 2005. A Taxonomy of Variability Realization Techniques. *Softw. Pract. Exper.* 35, 8 (2005), 705–754.

Feature Modeling vs. Decision Modeling: History, Comparison and Perspectives

Rick Rabiser
Christian Doppler Lab MEVSS, ISSE
Johannes Kepler University Linz, Austria
rick.rabiser@jku.at

ABSTRACT

Modeling variability, i.e., defining the commonalities and variability of reusable artifacts, is a central task of software product line engineering. Numerous variability modeling approaches have been proposed in the last three decades. Most of these approaches are based on feature modeling (FM) or decision modeling (DM), two classes of variability approaches that go back to initial proposals made in the early 1990ies, i.e., FODA for FM and Synthesis for DM. This extended abstract summarizes the history of FM and DM as well as the results of a systematic comparison between FM and DM published earlier. We also outline perspectives, especially regarding potential synergies and key common elements that should be part of a standard variability modeling language.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; *Designing software*; *Software development methods*; *Software development techniques*.

KEYWORDS

Variability modeling, feature modeling, decision modeling

ACM Reference Format:
Rick Rabiser. 2019. Feature Modeling vs. Decision Modeling: History, Comparison and Perspectives. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3307630.3342399

1 INTRODUCTION

Most existing FM approaches are more or less directly derived from the work on Feature-Oriented Domain Analysis (FODA) by Kang et al. [15]. DM exists nearly as long as FM, and, similarly, most (if not all) existing DM approaches have been influenced by one initial work, the Synthesis method [7]. Other approaches to variability modeling that have been proposed are, e.g., Orthogonal Variability Modeling (OVM) [17] and UML-based variability modeling [12]. Several works have focused on comparing existing variability modeling approaches and languages [10, 11, 18, 21] and also on analyzing their evaluation [6] and use in industry [4] as well as existing tool support [3].

In this paper, however, we focus solely on DM and FM. The author of this paper has been involved in two earlier efforts to compare DM approaches [20], and FM and DM approaches [8]. This paper summarizes these efforts and concludes with some perspectives towards developing a standard variability modeling language.

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342399

2 HISTORY

We briefly discuss the history of FM and DM.

2.1 Feature Modeling

FM was originally proposed as part of the FODA method [15]. In FODA, a domain is defined as a set of current and future systems sharing common capabilities. Domain analysis aims at discovering and representing commonalities and variabilities among them. In FODA, feature models capture features—*the end-user's (and customer's) understanding of the general capabilities of systems in the domain*—and the relationships among them.

FODA has inspired a multitude of works on extending the original notation and on modeling and implementing systems using feature models. Notable extensions include group cardinalities [19], feature cardinalities [9], and feature inheritance [2]. FM is also an integral part of feature-oriented software development (FOSD), a paradigm focused on treating features as modular, first-class entities throughout the entire development cycle [1]. Thus, the meaning of the term feature has been broadened dramatically over time.

2.2 Decision Modeling

The earliest documented approach to DM was proposed as part of the Synthesis method [7]. This method, developed by the Software Productivity Consortium for industrial use, provided an early reuse process model. Synthesis defines a decision model as *a set of decisions that are adequate to distinguish among the members of an application engineering product family and to guide adaptation of application engineering work products.* This early definition emphasizes product derivation—as opposed to describing the domain, which is the main focus of FODA [15].

Most DM approaches that were proposed after Synthesis (see our earlier study [20] for a detailed discussion) were either inspired by industrial applications or developed in close collaboration with industry.

3 COMPARISON

In our earlier work [8], we systematically compared FM and DM along *ten dimensions*: applications, unit of variability (feature vs.

Table 1: *Commonalities* (in italics) and differences (in bold) between FM and DM in diverse dimensions [8].

dimension	Feature Modeling	Decision Modeling
applications	**diverse applications**: **concept modeling (e.g., domain modeling)**, *variability* **and commonality** *modeling*; *derivation support*	*variability modeling*; *derivation support*
unit of variability	**features are properties of concepts, e.g., systems**	**decisions to be made in derivation**
orthogonality	mostly used in orthogonal fashion	orthogonal
data types	*comprehensive set of basic types*; **references**; *composite*: **via hierarchy, group and feature cardinalities**	*comprehensive set of basic types*; *composite*: **sets, records, arrays**
hierarchy	**essential concept; single approach**: tree hierarchy modeling, parent-child configuration constraints and **decomposition**	**secondary concept; diverse approaches**, e.g., **visibility or relevance** hierarchy (no decomposition)
dependencies & constraints	*no standard constraint language but similar range of approaches (Boolean, numeric, sets, quantifiers)*	*no standard constraint language but similar range of approaches (Boolean, numeric, sets)*
mapping to artifacts	**optional concept**; *no standard mechanism*	**essential concept**; *no standard mechanism*
binding time and mode	*not standardized, occasionally supported*	*not standardized, occasionally supported*
modularity	*no standard mechanism*; **feature hierarchy plays partly this role**	*no standard mechanism*; **decision groups play partly this role**
tool aspects	*representation of models as lists, tables, trees, and graphs*; **configuration UI: usually a tree (unordered)** *diverse solutions for supporting configuration workflows* (**secondary concept**)	*representation of models as lists, tables, trees, and graphs*; **configuration UI: an (ordered) list of questions** *diverse solutions for configuration workflows* (**essential**)

decision), orthogonality, data types, hierarchy, dependencies and constraints, mapping to artifacts, binding time and mode, modularity, and tool aspects. Table 1 summarizes the results of our comparison [8].

4 CONCLUSIONS AND PERSPECTIVES

In our earlier work [8] we not only compared FM and DM but also discussed variability modeling in practice. Specifically, using the ten dimensions, we also analyzed the Linux kernel variability specification language Kconfig [22], the eCos operating system's Component Description Language (CDL) [24], and the proposal of the Common Variability Language (CVL) standard [13], which eventually and unfortunately failed to become a standard.

From our own efforts [8, 20] as well as similar efforts [3–6, 10, 11, 18, 21, 23], we draw the following main conclusions regarding FM vs. DM, that should be considered when aiming to develop a common, simple, standard variability modeling language:

The main difference between FM and DM is that FM aims to support both commonality and variability modeling, whereas DM focuses exclusively on variability modeling. Specifically, *FM originally aimed to support domain analysis by modeling commonality and variability and DM originally focused on modeling variability to support application engineering/product derivation.* However, in practice FM is also often used to "just" model variability and often also with the goal to support product derivation or configuration. Furthermore, the constructs used to model variability can also be used to model commonality, thus allowing to model commonalities with DM too.

All other differences are either historical or only minor. Today, FM is mostly used, like DM, as an orthogonal variability modeling technique, where features, like decisions, abstract over variability pertaining to different types of properties (functional and non-functional), levels of abstraction (environment, system, subsystem) and life-cycle stages (requirements, design, implementation, test).

Noteworthy minor differences are: (i) *hierarchy* is essential and has uniform semantics (child-presence-to-parent-presence implications) in FM and is secondary and has varied semantics (child-to-parent-presence or child-to-parent-visibility implications) in DM; (ii) *mapping to artifacts* is essential in DM and, depending on the use case, optional in FM. In DM, while some decision-to-artifact mappings represent transformations, in other approaches they represent artifacts to include or to remove [20]. In FM, tools such as

FeatureMapper [14] use relationships among features and model elements to implement negative variability (i.e., by removing elements). Other tools such as FeatureIDE [16] have diverse "composers" with different internal representations. Each solution has its own pros and cons.

Key commonalities are: (i) both FM and DM have a similar range of *data types*; (ii) both provide similar constructs for expressing *constraints* and supporting *modularity*; and (iii) both lack standardized support for *binding time and mode*.

The main conclusion of our earlier study [8] still holds seven years later: *there is a significant convergence between FM and DM*. This is particularly evident when looking at practical variability modeling approaches, such as Kconfig and CDL: they combine concepts from FM and DM.

One should not focus too much on a particular instance of a FM or DM approach but learn from the whole set of available approaches, FM and DM, as well as other variability modeling approaches such as OVM and UML-based approaches. Our experiences suggest that a simple, standard variability modeling language should:

- support the typical basic data types known from programming languages and some type of composite,
- be orthogonal and independent of specific artifacts,
- provide a simple and clear concept to realize hierarchy and modularity,
- offer a simple and expressive way to define constraints and dependencies including mapping to concrete artifacts,
- support different use cases such as domain analysis or product configuration, but have a clear focus on the core use case: variability modeling,
- consider binding time and mode,
- and be as tool-independent as possible, i.e., allow using standard text editors as well as fully-fledged IDEs to define models.

ACKNOWLEDGMENTS

The financial support by the Christian Doppler Research Association, the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Primetals Technologies is gratefully acknowledged. We also explicitly want to thank the authors of the earlier study which was the main basis for this summary [8].

REFERENCES

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Development: Concepts and Implementation.* Springer.
[2] Timo Asikainen, Tomi Mannisto, and Timo Soininen. 2006. A unified conceptual foundation for feature modelling. In *Proceedings of the 10th International Software Product Line Conference*. IEEE, 31–40.
[3] 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), 14:1–14:45.
[4] 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 7th International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 7–14.
[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, 73–82.
[6] 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.
[7] Software Productivity Consortium. 1991. *Synthesis Guidebook*. Technical Report. SPC-91122-MC. Herndon, Virginia: Software Productivity Consortium.
[8] 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 6th International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 173–182.
[9] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing cardinality-based feature models and their specialization. *Software process: Improvement and practice* 10, 1 (2005), 7–29.
[10] 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, 12–21.
[11] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. 2013. Variability in software systems-a systematic literature review. *IEEE Transactions on Software Engineering* 40, 3 (2013), 282–306.
[12] Hassan Gomaa. 2005. *Designing software product lines with UML*. IEEE.
[13] Øystein Haugen, Andrzej Wąsowski, and Krzysztof Czarnecki. 2013. CVL: common variability language. In *Proceedings of the 17th International Software Product Line Conference*. ACM, 277–277.
[14] Florian Heidenreich, Jan Kopcsek, and Christian Wende. 2008. FeatureMapper: mapping features to models. In *Proceedings of the 30th International Conference on Software Engineering*. ACM, 943–944.
[15] 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.
[16] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.
[17] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Science & Business Media.
[18] 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.
[19] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. 2002. Extending feature diagrams with UML multiplicities. In *Proceedings of the 6th World Conference on Integrated Design & Process Technology*. Society for Design and Process Science, 1–7.
[20] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. 2011. A comparison of decision modeling approaches in product lines. In *Proceedings of the 5th International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 119–126.
[21] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature diagrams: A survey and a formal semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference*. IEEE, 139–148.
[22] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2010. The Variability Model of The Linux Kernel. In *Proceedings of the 5th International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 45–51.
[23] Marco Sinnema and Sybren Deelstra. 2007. Classifying variability modeling techniques. *Information and Software Technology* 49, 7 (2007), 717–739.
[24] Bart Veer and John Dallaway. 2011. The eCos Component Writer's Guide. *Manual, available online at http://www.gaisler.com/doc/ecos-2.0-cdl-guide-a4.pdf* (2011).

A Component-Based Approach to Feature Modelling

Pablo Parra
pablo.parra@uah.es
Space Research Group
University of Alcalá
Alcalá de Henares, Madrid, Spain

Óscar R. Polo
Space Research Group
University of Alcalá
Alcalá de Henares, Madrid, Spain

Segundo Esteban
ISCAR Research Group
Complutense University of Madrid
Madrid, Spain

Agustín Martínez
Space Research Group
University of Alcalá
Alcalá de Henares, Madrid, Spain

Sebastián Sánchez
Space Research Group
University of Alcalá
Alcalá de Henares, Madrid, Spain

ABSTRACT

This paper presents an approach to feature modelling based on the use of modelling constructs from the component-based software development domain. The proposed models allow establishing feature hierarchies, making a clear distinction between the features themselves and their realisations or variants. Furthermore, they enable the definition of complex dependency relationships between the different feature realisations, making it possible to define variable configurations associated with these dependencies. Finally, the approach allows the modelling of product configurations as a set of interconnected and configured feature realisations. The proposal is illustrated with an example based on the on-board satellite software applications domain.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Abstraction, modeling and modularity**; Model-driven software engineering.

KEYWORDS

software product lines, variability, feature models

ACM Reference Format:
Pablo Parra, Óscar R. Polo, Segundo Esteban, Agustín Martínez, and Sebastián Sánchez. 2019. A Component-Based Approach to Feature Modelling. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3307630.3342402

1 INTRODUCTION

Product line software engineering [3] is a methodology aimed to reduce both costs and time when developing families of software products within a specific domain, i.e. products that share a common structure or functionality but that present a certain degree of variability from one to another.

One of the key concepts of software product lines is variability, understood as the "commonalities and differences in the applications in terms of requirements, architecture, components and test artifacts" [9]. There are several approaches to expressing variability, including Feature Modelling (FM) [7], Decision Modelling (DM) [12] and Orthogonal Variability Management (OVM) [9].

Feature models express variability in terms of product features, i.e. prominent or distinctive user-visible aspects, qualities or characteristics of software systems [6]. Many of the FM approaches are based or derived from the work on Feature-Oriented Domain Analysis (FODA) [6], and traditionally use tree-based feature diagrams to specify the members of a product line. The original notation has been extended several times with multiple capabilities [1, 2, 4, 10, 13].

This paper presents an approach to feature modeling inspired by the artifacts characteristic of component-based software design. The defined models allow establishing feature hierarchies, clearly differentiating between the features themselves and their variants or realisations. Furthermore, it enables the definition of complex dependency relationships between the different feature realisations, making it possible to define variable configurations associated with these dependencies. Finally, the approach allows the modelling of product configurations as a set of interconnected and configured feature realisations.

The on-board satellite software domain has been used as an example of the proposed approach. This scope of work constitutes one of the main areas of expertise of the authors and will provide examples for the different model elements.

The rest of the paper is organised as follows: the following section provides a general description of the On-Board Software domain, which will be the source of examples of the different modelling elements. Section 3 defines the constituent constructs of the component-based feature modeling approach. Finally, the final section contains conclusions and future work.

2 ON-BOARD SOFTWARE DOMAIN

The OBSW [11] of a satellite is in charge of controlling the main overall procedures that take place in the spacecraft. These are, among others, managing the transmission of information between

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342402

the spacecraft and the ground stations, reporting the state of the spacecraft by performing a set of so-called *housekeeping* operations, performing monitoring and control cycles on certain variables, such as the temperature of the different devices or the orientation angles of the solar panels, and controlling the execution of the different processes and functions performed by the payloads present in the spacecraft. Nevertheless, depending on the type of satellite, e.g. scientific, communication, etc., the overall procedures that the OBSW might perform may vary.

Conceptually, an OBSW can be divided into *applications*. These applications are software products that have their own specification and validation procedures. Each one is in charge of one aspect of the system, for example, thermal control or attitude and orbit control systems (AOCS). The main actions executed by them are, in the vast majority of cases, fulfilled by periodic tasks running cyclically at a given frequency in order to ensure a stable and accurate execution pattern.

Applications can be commanded by the ground station or by other applications running in the satellite. They can receive orders in the form of telecommand packets that will request the execution of a given action and, depending on the functionality they implement, produce telemetry information to be delivered to ground. This telemetry can include, for example, information regarding its state or the result of executing a particular command.

There is also an event service that allows the reporting of events between the different applications. This service can be used to send notifications of certain events that might happen in the course of the applications' execution, such as a failure or the occurrence of a given phenomenon external or internal to the spacecraft, and that might trigger further actions that should be taken by any of the other applications.

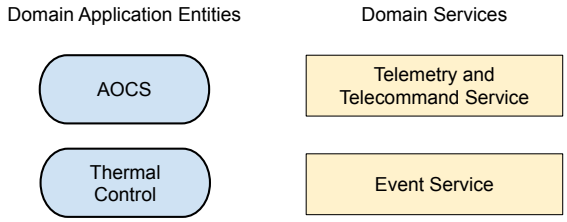


Figure 1: Key elements of the OBSW product domain

Figure 1 shows a diagram with the constituent elements of the OBSW product domain that will be used as an example in the following section.

3 COMPONENT-BASED FEATURE MODELLING

The proposed solution is based on the use of modelling constructs from the component-based software development domain to model variability in software product lines. More specifically, the constructs and artefacts used in this approach have a direct relationship with the component meta-model introduced in [8].

The core element of the model is the *feature*. A feature is a prominent or distinctive characteristic of a software system that is susceptible to having different realisations or variants. In our approach, features are modelled as classifiers, taking as equivalent construct the component type element of the component meta-model. These classifiers establish the characteristics associated with each feature. The complete set of features determines the *domain* of the software product line.

Once the features of a given domain have been established, feature realisations can be defined. A *feature realisation* or *feature variant* is each one of the possible alternatives of implementing a feature. In our approach, features are modelled similarly to components. These realisations can define or instantiate points of interaction, that is, points through which feature realisations can request or provide functionalities or services to other feature realisations. These interaction points, also called *ports* following the terminology used in the component model domain, can have two different roles: *client* and *server*. By means of client ports, a feature realisation declares the services it needs in order to be configured properly. Conversely, a feature realisation can provide a certain service through a server port. Every port is associated with an *interface* that configures the service that is being provided and required.

Finally, feature realisations can be deployed into *product configurations* which are defined as a set of interconnected and configured feature realisations that model a single product within the software product line. The connections that establish the interactions between feature realisations are always between one or more client ports and a server port. The following subsections detail each of the elements of the proposed model. Examples directly related to the on-board software domain presented in the previous section are shown for each of the elements.

3.1 Product Features

As previously mentioned, product features are modelled in the form of classifiers which establish the fundamental characteristics that must meet all their realisations or variants. Features are declared as part of a product line domain, whose meta-model is shown in Figure 2.

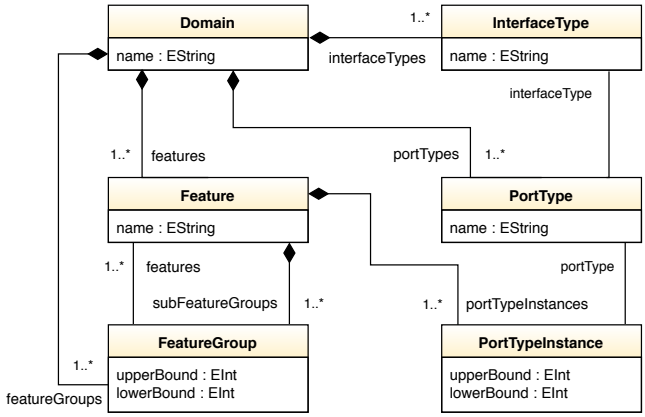


Figure 2: Product Domain Meta-Model

A domain, represented in the meta-model by the class `Domain`, includes the definition of one or more features (`Feature`), port types

(PortType) and interfaces (InterfaceType). Port types are used to differentiate the type of interaction expected between two or more feature realisations. Each port type has an associated interface type. The interface type establishes the mechanisms necessary to perform the configuration of the interaction. These mechanisms can be, for example, a specific meta-model or an Interface Description Language (IDL).

The class FeatureGroup can be used to define feature groups within a domain. A *feature group* is a set of features annotated with a cardinality specifying how many feature realisations can be deployed and configured from that set in a given product. This cardinality-based mechanism, similar to the one described in [5], enables the definition of alternative groups, such as *inclusive-or* and *exclusive-or* groups. Furthermore, the meta-model also allows establishing a feature hierarchy by defining groups of subfeatures within a given feature. These subfeatures are modelled as if they were an internal part of the containing feature. In this way, it is possible to encapsulate the information corresponding to the subfeatures of each realisation, hiding the details between the different levels of the hierarchy and thus reducing the complexity of the models.

Besides establishing the cardinality of feature realisations within a product configuration, the meta-model also allows restricting, through the class PortInstance, the cardinality, type and role of the ports they can deploy.

To define the domain model of a software product line, it is possible to use a textual syntax like the one that appears in Listing 1. This example corresponds to the definition of the OBSW domain described in the previous section.

Two main features are defined: Applications and Services, each forming part of a specific feature group with cardinality one. Thus, all product configurations must define one realisation of each feature. The first feature encapsulates all the applications defined by the embedded software, while the second groups all the services that the applications will have to use during their execution processes. As it is observed, within Applications, a set of subfeature groups are defined with cardinality one, specifically one for each type of application, that is, AOCS and ThermalControl. The internal part of the feature Services defines the features corresponding to the telemetry and telecommand (TMTCService) and event (EventService) services. To indicate that the last subfeature is optional, its cardinality ranges between zero and one.

In addition to the features, the domain also establishes a set of port types: TMTCPortType, which models the interaction between the applications and the telemetry and telecommand service; and EvtServicePortType for the interactions between the applications and the event service. Each of them has an associated interface type that will allow, at the moment the feature is realised, the specific configuration of each interaction.

A feature defines the cardinality of the ports that its realisations can instantiate. In this case, the feature realisations corresponding to the services, that is, TMTCService and EventService, shall define a server port of type TMTCPortType and EvtServicePortType, respectively. The feature realisations associated with the applications have to define a client port of type TMTCPortType and may or may not define a client port of type EvtServicePortType. Through these client ports, the applications will be able to configure the services they require. The root features, i.e., Applications and Services, enable the definition of internal and external ports.

Listing 1 Definition of the OBSW domain

```
domain OBSW {
  interface types {
    interface type TMTCInfaceType;
    interface type EvtServiceIfaceType;
  };
  port types {
    port type TMTCPortType uses TMTCInfaceType;
    port type EvtServicePortType uses EvtServiceIfaceType;
  };
  features {
    feature AOCS {
      port instances {
        external client TMTCPortType range 1 to 1;
        external client EvtServicePortType range 0 to 1;
      };
    };
    feature ThermalControl {
      port instances {
        external client TMTCPortType range 1 to 1;
        external client EvtServicePortType range 0 to 1;
      };
    };
    feature TMTCService {
      port instances {
        external server TMTCPortType range 1 to 1;
      };
    };
    feature EventService {
      port instances {
        external server EvtServicePortType range 1 to 1;
      };
    };
    feature Services {
       port instances {
         internal client TMTCPortType range 1 to 1;
         internal client EvtServicePortType range 0 to 1;
         external server TMTCPortType range 1 to 1;
         external server EvtServicePortType range 0 to 1;
       };
       subfeature groups {
         group [ TMTCService ] range 1 to 1;
         group [ EventService ] range 0 to 1;
       };

    feature Applications {
      port instances {
        internal server TMTCPortType range 2 to 2;
        internal server EvtServicePortType range 0 to 2;
        external client TMTCPortType range 2 to 2;
        external client EvtServicePortType range 0 to 2;
      };
      subfeature groups {
        group [ AOCS ] range 1 to 1;
        group [ ThermalControl ] range 1 to 1;
      };
    };
  };
  feature groups {
    group [ Applications ] range 1 to 1;
    group [ Services ] range 1 to 1;
  };
};
```

These ports are used to publish internal dependencies externally, and their semantics are described in the following subsection.

3.2 Feature Realisations

Once the features of a domain have been established, the model allows defining the different feature realisations, that is, the different implementations or variants of each feature that can be used to build a product configuration. Feature realisations will always have an associated feature and will be able to define one or more ports depending on the services they are going to provide or demand, always according to the definition provided by the feature domain. The simplified feature realisations meta-model is shown in Figure 3.

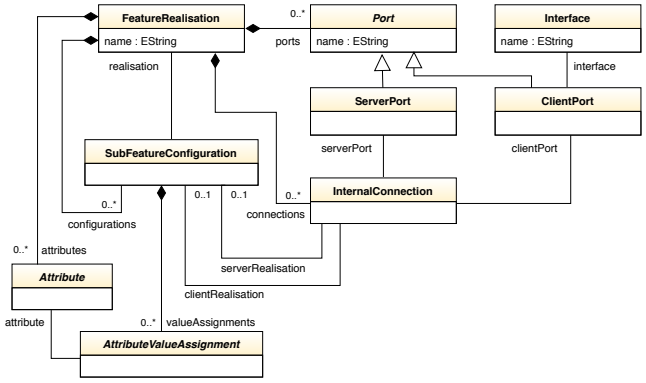


Figure 3: Feature Realisation Meta-Model

A feature realisation can define a set of attributes, the final value of which must be set when deploying and configuring the realisation, either as part of a product configuration or as an internal part of another feature realisation of a higher hierarchical level. Attributes add one more variability point to the definition of feature realisations, since it allows the association of different values in each deployment of a product configuration

Feature realisations define a set of ports whose cardinality is defined by the feature they implement. As already described, ports can have two roles, i.e. client (`ClientPort`) and server (`ServerPort`) and are associated with a certain port type. In addition, the ports in turn can be internal or external, that is, they can define an interaction with an external feature realisation, or with an internal subfeature.

Listing 2 shows the definition of the `PUSTMTCService` feature realisation. This variant uses the Packet Usage Standard (PUS) defined by the European Space Agency for its space missions. It defines an external server port of type `TMTCPortType` through which different applications can declare, via the interfaces of their client ports, which PUS services and subservices they are going to use. In addition, it establishes as an attribute the maximum number of packets per second that the system is able to process.

Client ports instantiated as part of a feature realisation must declare an interface. This interface, the type of which must match that associated with the port type, allows the configuration and modelling of the services demanded. The way in which this modelling is carried out will depend on the type of interface. As mentioned

Listing 2 `PUSTMTCService` feature realisation

```
realisation PUSTMTCService of TMTCService {
    attributes {
        integer MAX_PACKETS_PER_SEC;
    };
    ports {
        external server TMTCPortType tmtcServer;
    };
};
```

in the previous subsection, the interface types can have associated in each case a meta-model or a description language that allows configuring the interaction. Listing 3 shows an example of a description language defined for the type of interface corresponding to the telemetry and telecommand service. Using this language, applications can define and parameterize the telemetry messages they generate and the telecommands they accept.

Listing 3 Example of the `TMTCInterfaceType` IDL

```
TMTCInterface TMTCThermalApp {
    telecommands {
        telecommand EnableHK {
            serviceID := HKAndDiagnostic;
            subserviceID := 0;
        };
        telecommand DisableHK {
            serviceID := HKAndDiagnostic;
            subserviceID := 1;
        };
    };
    telemetries {
        telemetry HKReport {
            serviceID := HKAndDiagnostic;
            subserviceID := 25;
            parameters {
                float32 tempLineA;
                float32 tempLineB;
                float32 tempLineC;
            };
        };
    };
};
```

A feature realisation may include, within its definition, deployment and configuration of one or more subfeature realisations, represented by the class `SubFeatureConfigurations`. These configurations model the chosen variants for each of the possible internal subfeatures established in the domain model. When a feature realisation is displayed, it is necessary to assign values to the possible attributes that may have been included in its definition.

To define an interaction between two configured feature realisations it is necessary to establish a connection between the corresponding ports of each of them. In the case of subfeature configurations, these connections, modelled through the `InternalConnection` class, can be made between two subfeature configurations or between the main realisation and its subfeatures. A well-defined connection must always be established between a client port and a server port of the same type. The meta-model also allows setting up port relays. A relay type connection is one established between

an internal port and an external port of the same feature realisation. Through these special connections, feature realisations can externally publish internal ports. In this way, direct interactions can be established between the internal subfeatures and the external feature realisations.

Listing 4 `EOSatelliteServices` feature realisation

```
realisation EOSatelliteServices of Services {
  ports {
    internal client TMTCPortType tmtcRelay;
    external server TMTCPortType tmtcServer;
    internal client EvtServicePortType evtRelay;
    external server EvtServicePortType evtServer;
  };
  subfeature configurations {
    configuration PUSTMTCService pusTMTCService {
      MAX_PACKETS_PER_SEC := 10;
    };
    configuration PUSEventService pusEventService { };
  };
  connections {
    connection this.tmtcRelay <-> this.tmtcServer;
    connection this.evtRelay <-> this.evtServer;
    connection this.tmtcRelay <->
        pusTMTCService.tmtcServer;
    connection this.evtRelay <->
        pusEventService.evtServer;
  };
};
```

Listing 4 shows the realisation of the feature `Services` for an Earth Observation satellite. In this example, the realisation defines two external server ports, `tmtcServer` and `evtServer` through which the telemetry and telecommand and event services will be provided to the applications. These ports are in turn connected to the internal configurations `pusTMTCService` and `pusEventService` using relay ports. This models the fact that the features that actually provide the services are the internal ones, and that their points of interaction are published externally by means of equivalent ports of the containing feature realisation. Finally, when configuring the internal feature `pusTMTCService`, a value is assigned to the attribute `MAX_PACKETS_PER_SEC`, defined by the corresponding feature realisation.

3.3 Product Configurations

Once the feature realisations have been defined, the last step is to build a product configuration by deploying, configuring and interconnecting feature realisations. The definition will be adjusted to the domain definition of the software product line. The meta-model of a product configuration is shown in Figure 4.

A product configuration defines a set of feature configurations, represented by the class `FeatureConfiguration`. This configuration select the variants chosen for each of the root features defined in the domain model. These configurations must include the value assignments to the eventual attributes defined by the corresponding realisations.

Once the features have been configured, the product is completed by defining the connections between the different ports. These connections, modelled using the class `Connection`, are always between a client port of one feature and a server port of another feature.

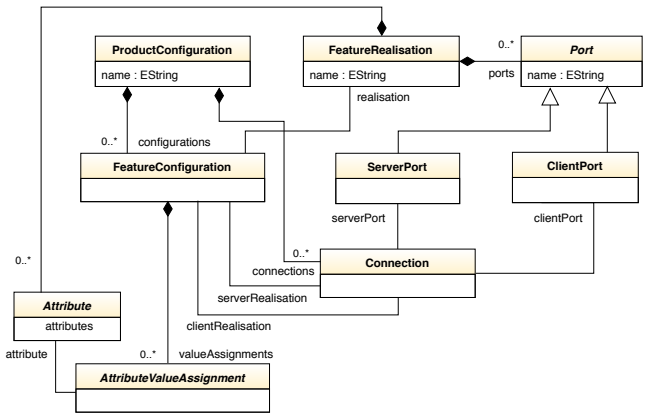


Figure 4: Product Configuration Meta-Model

Listing 5 `EOSatellite` product configuration

```
product EOSatellite {
  configurations {
    configuration EOSApplications eosApplications { };
    configuration EOSServices eosServices { };
  };
  connections {
    connection eosApplications.tmtcClient <->
        eosServices.tmtcServer;
    connection eosApplications.evtClient <->
        eosServices.evtServer;
  };
};
```

Listing 5 defines the configuration of the Earth Observation satellite product `EOSatellite`. In accordance with the OBSW domain, only two feature configurations shall be defined: one that realises the feature `Applications` and the other that does the same with the feature `Services`. The product configuration shall also establish the connections between the ports in such a way that all the interactions are correctly defined. In this case, the client ports of `EOSApplications` are connected to the corresponding ports of the `EOSServices` configuration.

4 CONCLUSIONS AND FUTURE WORKS

This paper has introduced an approach to feature modelling based on the use of constructs from the component-based software development domain. The proposed models allow establishing features hierarchies, making a clear distinction between the feature themselves and their realisations or variants. Furthermore, it enables the definition of complex dependency relationships between the different feature realisations, making it possible to define variable configurations associated with these dependencies. Finally, the approach allows the modelling of product configurations as a set of interconnected and configured feature realisations. One of the future goals of this work is to define a model-based software product line of on-board satellite applications that, using as inputs the feature models defined in this approach, allows building products with

a high degree of asset reuse and complete control of the configurations needed to meet the requirements of each application.

5 ACKNOWLEDGMENTS

This work has been supported by Spanish Ministerio de Economía y Competitividad under the grant ESP2017-88436-R.

REFERENCES

[1] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. of the International Conference on Software Product Lines (SPLC) 2005*. Springer, Berlin, Heidelberg, 7–20. https://doi.org/10.1007/11554844_3

[2] 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*. Springer-Verlag, 491–503. https://doi.org/10.1007/11431855_34

[3] Paul Clemens and Linda Northrop. 2001. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[4] Krzysztof Czarnecki, Michal Antkiewicz, Chang Hwan Peter Kim, Sean Lau, and Krzysztof Pietroszek. 2005. Model-driven software product lines. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*. ACM, New York, NY, USA, 126–127.

[5] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2004. Staged Configuration Using Feature Models. In *Software Product Lines*, Robert L Nord (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 266–283.

[6] 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. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231

[7] K.C. Kang, Jaejoon Lee, and P. Donohoe. 2002. Feature-oriented product line engineering. *IEEE Software* 19, 4 (jul 2002), 58–65. https://doi.org/10.1109/MS.2002.1020288

[8] Pablo Parra, Oscar R. Polo, Martin Knoblauch, Ignacio Garcia, and Sebastian Sanchez. 2011. MICOBS: multi-platform multi-model component based software development framework. In *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering (CBSE '11)*. ACM, New York, NY, USA, 1–10.

[9] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/3-540-28901-1

[10] Matthias Riebisch, Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. 2002. Extending Feature Diagrams With UML Multiplicities. In *Proc. of the 6th World Conference on Integrated Design and Process Technology*. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.1653

[11] Ana-Elena Rugina. 2010. *Defintion and Characterization of On-Board Software Concepts. Towards Component-Based Software Engineering*. Technical Report. EADS Astrium SAS.

[12] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. 2011. A comparison of decision modeling approaches in product lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems - VaMoS '11*. ACM Press, New York, New York, USA, 119–126. https://doi.org/10.1145/1944892.1944907

[13] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)*. IEEE, 139–148. https://doi.org/10.1109/RE.2006.23

Answering the Call of the Wild?: Thoughts on the Elusive Quest for Ecological Validity in Variability Modeling

Seiede Reyhane Kamali
École de technologie supérieure, Université du Québec
Montreal, Canada
seiede-reyhane.kamali.1@ens.etsmtl.ca

Shirin Kasaei
École de technologie supérieure, Université du Québec
Montreal, Canada
shirin.kasaei.1@ens.etsmtl.ca

Roberto E. Lopez-Herrejon
École de technologie supérieure, Université du Québec
Montreal, Canada
roberto.lopez@etsmtl.ca

ABSTRACT

Ecological validity is a term commonly used in several disciplines to refer to the fact that in a research study, the methods, the materials, and the settings must approximate the real world, i.e. what happens in everyday life. Variability modeling is no exception, it has striven for this form of validity by looking at two main sources, industrial projects and open source projects. Despite their unquestionable value, industrial projects inherently pose limitations; for instance, in terms of open access or results replication, which are two important tenets for any scientific endeavor. In this paper, we present our first findings on the use of open source projects in variability modeling research, and identify trends and avenues for further research.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

feature models, variability models, open source projects

ACM Reference Format:
Seiede Reyhane Kamali, Shirin Kasaei, and Roberto E. Lopez-Herrejon. 2019. Answering the Call of the Wild?: Thoughts on the Elusive Quest for Ecological Validity in Variability Modeling . In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3307630.3342400

1 INTRODUCTION

Every empirical study needs to consider any threats to validity that might occur from the planning of an experiment to the analysis of its results. In software engineering, different classifications of threats to validity have been proposed and employed. A commonly referred one is the classification proposed by Wholin et al. [53]. In this classification, a threat to external validity refers to any conditions that *limit the ability to generalize the results of an experiment to industrial practice* [53]. Furthermore, a specific form of external validity threat, as identified in Wholin et al.'s work, is *interaction of setting and treatment* which is defined as not having the experimental setting or artifacts that resemble those of a real-life context of interest such as industrial practice.

Ecological validity is a more precise term for describing this particular form of external validity and it is used extensively in social and medical sciences (e.g. [12]). In a nutshell, ecological validity asks the question: is a given study, its methods, its artifacts and its setting an approximation to what happens in everyday life, a.k.a. *"the real world"* or *"the wild"*? Clearly, ecological validity has been a major and driving concern in *software product lines* (SPL) research and in particular for variability modeling. Both industrial projects and open source projects have been the main sources to address ecological validity. A wealth of industrial research experience has been collected over the last two decades (e.g. [44]); however, because of its nature, industrial projects pose limitations in terms of open access to methods, artifacts or settings. To circumvent these issues, the alternative of using open source projects has been proposed and extensively explored during the last decade.

In this paper, we present our first results in analyzing the impact of open projects in SPL research with the particular focus on feature models. We analyzed what open projects have been used, for what purposes, how recent their extracted feature models are and in what format they are represented, what development activities they cover, and if their artifacts are available for replication. We believe our work will help raise awareness of both the pros and the pitfalls of using open source projects as a source for ecological validity, that can inform the design decisions for the next generation of variability models.

2 STUDY SET UP

In this section, we first describe the process that we followed to identify the open source projects that have been reported in literature for variability modeling. We present and analyze the most salient characteristics of those projects, and enumerate the threats to validity of our study and how we addressed them.

2.1 Study description

There exists a vast body of knowledge regarding variability modeling that has accumulated over more than two decades of research as attested in several literature reviews and surveys (e.g. [7] or [24]). Thus rather than beginning anew, our starting point was a recent survey performed by Galindo et al. [19]. In this survey, the authors analyzed papers relating to automated analysis of feature models

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342400

published between 2010 and 2017. At a first stage they identified, among other things, favored publication outlets. At a second stage, they performed a more in depth bibliometric analysis. In the latter stage, they obtained 242 primary sources that met one of the two following criteria. First, that the primary source appeared in a publication outlet (identified in their first stage) with a high acceptance of papers on variability modeling and automated analysis of feature models. Some examples of such outlets are the Variability Modelling of Software-Intensive Systems (VaMoS), the International Systems and Software Product Line Conference (SPLC), the International Conference on Automated Software Engineering (ASE), or the International Conference on Software Engineering (ICSE). Second, that the primary source was among the top 10 percentiles according to SciVal[1] from the papers identified in their first stage. For further details on their selection process and results, please refer to [19]. We used these 242 papers for performing our study.

First and foremost, we must characterize our use of the term *open source project*. For our study, we considered an open source project the project that has a publicly available code repository (loosely interpreted as a URL to obtain the source code and related software artifacts), and with more than one contributor (authors and committers). It is important to indicate that these two criteria are less stringent than, for instance, the definition provided by the Open Source Initiative[2] or some of the guidelines by Kalliamvakou et al. in their seminal work on GitHub mining [27]. We chose these criteria at this stage of our work to potentially canvas a broader spectrum of open source projects. We should also point out that we excluded from our study projects that have URLs, for instance those collected in SPL2GO[3], which are fundamentally academic small-size projects.

The application of our criteria yielded 41 papers from which we collected the details of the projects such as name, versions, number of features, and formats of the feature models. In addition, we searched the number of project versions elapsed to date from the last one reported on the primary sources with the objective of getting a sense of how up-to-date are the versions that have been analyzed. Furthermore, we extracted the SPL development subactivities where the feature models were used for and the availability of the code sources, the related tools and artefacts to replicate the work described in the papers, and the formats used to store the feature models. This information allows us to measure the interplay that feature models have with other artifacts at different sub-activities of SPL development. The data was independently collected by the three authors of the paper after piloting a few of these papers for calibration. Afterwards a series of meetings took place to discuss the information collected individually until a consensus was reached. The next section summarizes the results obtained.

2.2 Results

Figure 1 shows the number of primary sources plotted against publication year. The histogram reflects no clear pattern, with highs in years 2012 and 2017 but otherwise stable with a median of 5 publications per year.

[1]https://scival.com/
[2]https://opensource.org/
[3]http://spl2go.cs.ovgu.de/projects

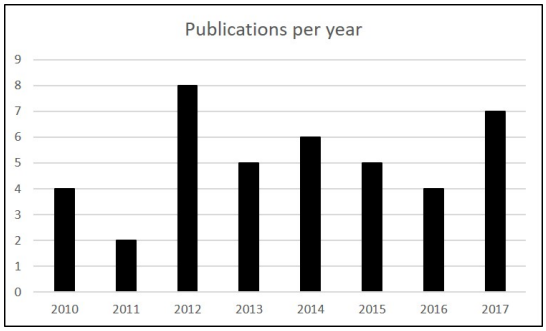


Figure 1: Histogram of publications per year

Table 1 and Table 2 summarize the projects we identified in the 41 primary sources of our study. We devoted its own table for Linux as it is the most studied open source project. In total, we found 43 distinct projects. In these tables, we show their names, the version numbers that have been studied, the number of features as reported, and the primary sources per each project and version. Notice that for each primary source listed per version row, we show the corresponding number of features in the same order. When the number of features was not reported, we used the token NA (Not Available). Similarly, when a version of a project was not reported we also used the same token NA. For sake of space, in the last row of Table 2 we present all the projects from von Rhein et al. for which neither the versions nor the number of features were reported [52].

Table 1 attests that, undoubtedly, Linux is by far the most frequently studied open source project with 25 out of the 41 primary sources using it as a case study. In total, 43 concrete distinct versions were reported, with 4 primary sources not reporting the versions used (entry NA). The earliest version reported was 2.6.12 and the most recent was 4.4.1. We noticed several salient facts. First, the

Table 1: Linux project summary

Versions	Features	Primary sources
(16) 2.6.12 to 2.6.27	3284 to 6319	[35]
2.6.28	3284 to 6319 ,6888	[35] [47]
2.6.28.6	5701, 5323, 6888, 6888, NA, 6888, 6888, 6888, 5321, 5426	[4] [8] [20] [21] [23] [22] [26][29] [49] [48]
(3) 2.6.29 to 2.6.31	3284 to 6319	[35]
2.6.32	6320, 6320, 3284 to 6319	[9] [10] [35]
2.6.32-2var	60072	[29]
2.6.33.3	6918, 6559, 6559, NA	[30] [36] [37] [52]
2.6.33.3-2var	62482	[29]
2.6.38	NA	[15]
(5) 2.6.39 to 3.3	NA, NA	[15] [16]
3.4	NA, NA, NA	[15] [16] [52]
(4) 3.5 to 3.8	NA, NA	[15] [16]
(6) 3.9 to 3.14	NA	[16]
4.4.1	63914	[18]
NA	NA, 5913, 6888	[1] [5] [54]

Table 2: Open source projects summary

Project	Versions	Features	Primary sources
BusyBox	1.18.0	881, 6796	[10] [29]
	1.18.5	792	[30]
	1.21.0	921, 921	[36] [37]
	NA	801, 881, NA	[5] [40] [52]
uClibc	0.9.31	369	[10]
	0.9.33.2	367, 367	[36] [37]
	NA	165, 369	[5] [40]
Eclipse	3.0	251	[42]
	(6) 3.1 to 3.6	177 to 621	[41] [42]
uClinux	20100825	(383, 1620) , 1850, 1850	[10] [20] [47]
	NA	(1850, 11254), 6888	[29] [54]
Drupal	7.22, 7.23	48, 28, 48	[39] [45] [46]
	NA	48	[25]
axTLS	1.2.7	108, 684	[10] [47]
	NA	684, 108	[29] [40]
Fiasco	2011081207	171, 1638, 1638	[10] [20] [47]
	NA	1638, 171	[29] [40]
BuildRoot	2010.11	1938	[10]
	NA	14910	[29]
FreeBSD	8.0.0	1203, 1396, 1396, 1396, 1396, 1203	[8] [20] [21] [22] [47] [49]
	NA	1396, 1396, NA	[26] [29] [52]
FraSCAti	(3) 1.3 to 1.5	50, 53, 60	[3]
	NA	89	[41]
EmbToolkit	0.1.0-rc12	1357	[10]
	NA	23516	[29]
Freetz	1.1.3	3471	[10]
	NA	31012	[29]
ToyBox	0.1.0	71, 544	[10] [47]
	NA	544	[29]
coreboot	4.0	2269	[10]
	NA	12268	[29]
eCos	3.0	1244, 1256, 1244, 1244, 1244, 1254, 1254, 1244, 1245	[9] [10] [20] [21] [22] [37] [36] [47] [49]
	NA	649 , 1244 , 1244 , 1244	[5] [54] [26] [29]
Ubuntu	10.04	7065, 7098, 8122, 26338	[38]
OpenSSL	1.0.1c	589	[30]
JDK Buffer library	1.5	73	[14]
JavaGeom	NA	110	[51]
ArgoUml	NA	11, 11, 11	[6] [31] [55]
BerkeleyDB	NA	42, 13, (18, 32), NA, 94	[11] [25] [50] [52] [4]
LLVM	NA	NA, 11	[52] [50]
SQLite	NA	39	[52] [50]
Gantt	NA	17	[34]
Apache http server	NA	NA, 9	[52] [50]
x264	NA	16	[50]
Cherokee, Gnuplot, Parrot, Postgresql, OpenVPN, Libxml2, Sendmail, Vim73, Xterm, Qemu, Xfig, Elevator, Gimp, Gnumeric, H264, Subversion			[52]

most popular version, 2.6.28.6, was reported with a wide range of number of features from 5321 (She et al.[49]) to the highest and most frequent number of 6888 features (e.g. Henard et al.[20]). Second, we identified three primary sources that studied the evolution of the variability model of Linux: Lotufo et. al who analyzed 21 versions [35], Dintzner et al. who first considered 10 versions (see [15]) and later expanded their scope to 16 versions [16]. Third, there is a

huge range in the number of features across all studied versions, going from 62482 features (see [29]) to 3284 (see [35]).

The second most frequently studied project was eCos with 13 primary sources that used it as a case study. In stark contrast with Linux, just a single concrete version is used, and the number of features reported go from 649 to 1256. In third place was FreeBSD with nine primary sources. Here again, one single version 8.0.0 was used and three primary sources did not report a version. In fourth place was BusyBox with eight primary sources, three of which did not report the versions used. The fifth place was tied with five primary sources for project uClibc, uClinux, and BerkeleyDB. The first with two concrete reported versions, the second with one reported version, and the last with no concrete reported version.

In terms of number of versions, the most diverse project is naturally Linux as mentioned before. The second place was Eclipse with seven versions, followed by BusyBox and FraSCAti with three concrete versions each, and Drupal and uClibc with two concrete versions each. From the rest of the projects, 12 reported a single concrete version whereas 8 did not report a concrete version (i.e. token NA in their only row). It should be noted also the paper by von Rhein et al. (see [52]) used 23 open source projects for which only the Linux project has concrete version numbers provided.

A second aspect that we wanted to study about the use of open source projects for variability modeling is their recency. In other words, how recent are the project versions that were analyzed in the primary sources in relation to the projects' versions releases. This information can help us gauge the time gap between the reality studied and that at the present. For this aspect, we searched during the first two weeks of May 2019 the most current release of each project. Granted, a caveat of our analysis is the fact that we considered primary sources up to 2017. Nonetheless, this choice permits us to gather interesting insights as described shortly.

Table 3 summarizes our findings regarding recency. This table contains the following information: *i)* the name of the project, *ii)* the year and month of the most recent version released, *iii)* the year and month of the most recent version studied among all the primary sources that analyzed the project, *iv)* the number of distinct concrete versions analyzed and optionally in parentheses the number of primary sources that did not report a version (entry NA in Table 1 and Table 2), *v)* the number of versions elapsed between the most recent released one and the most recent one analyzed (i.e. those versions corresponding to the second and third columns), and *vi)* the number of months elapsed between the most recent released one and the most recent one analyzed (again the versions corresponding to the second and third columns).

For sake of space, in Table 3, we omitted 17 projects for which there were no concrete version provided, thus it was not possible to ascertain their recency. This table is sorted in reverse chronological order by the year and month of the most recent version released in the second column.

Once more, Linux appears as the most prominent project with: a recent release, a recent version used in relation to its primary sources publication dates, 43 concrete versions studied and 4 primary sources not reporting a version, 18 versions elapsed within a period of 40 months. Interestingly, this project has the smallest gap even though 40 months is a period wider than the cut off date of year 2017 for our study — a gap of 17 months to May 2019.

Table 3: Open source projects recency summary

Project	Version Dates		Number of Versions		Time Elapsed months
	Recent	Last	Studied	Elapsed	
Linux	2019-05	2016-01	43(4)	18	40
Drupal	2019-05	2013-08	2(1)	198	69
BuildRoot	2019-05	2010-11	1(1)	178	102
x264	2019-05	2010-08	0(1)	77	105
Ubuntu	2019-04	2010-04	1	18	108
axTLS	2019-03	2010-08	1(2)	27	103
Eclipse	2019-03	2010-06	7	11	105
JDK buffer library	2019-03	2004-09	1	7	174
BusyBox	2019-02	2013-01	3(3)	30	73
ToyBox	2019-02	2009-12	1(1)	30	98
OpenSSL	2019-02	2012-05	1	3	81
FreeBSD	2018-12	2009-11	1(3)	18	97
Freetz	2018-12	2010-04	1(1)	13	104
coreboot	2018-12	2010-02	1(1)	9	106
Fiasco	2018-11	-	1(2)	-	-
EmbToolkit	2017-06	2010-08	1(1)	13	82
Parrot	2016-02	-	0(1)	-	-
Sendmail	2015-07	-	0(1)	-	-
JavaGeom	2014-02	2012-09	0(1)	1	17
Elevator	2013-05	-	0(1)	-	-
Cherokee	2014-04	-	0(1)	-	-
uClibc	2012-05	2012-05	2(2)	0	-
ArgoUML	2009-05	-	0(1)	-	-
eCos	2009-03	2009-03	1(4)	0	-
uClinux	-	2010-08	1(2)	-	-
FraSCAti	-	-	3(1)	-	-

Recent date of most recent version, *Last* date of most recent reported version, *Studied* versions studied in articles with (N) number of versions Not Available in Table 1 or Table 2, *Elapsed* number of versions elapsed from the last reported to the most recent.

There are several salient facts that can be distilled from this table. For instance, from the eleven projects with a release in 2019, the median of the time elapsed is 102 months. Projects such as Ubuntu, JDK buffer library, ToyBox, and FreeBSD have the last released prior to 2010, our lower bound cut off year of our primary sources. Furthermore, these projects exhibit a wide range of number of versions elapsed, from a high of 198 versions for Drupal to the case of eCos with only a single concrete version reported and hence no elapsed versions.

The bottom of the table shows the projects whose latest update is on or before 2017. For these projects, it must be highlighted that they used rather old versions in their studies. For example, EmbToolkit with a gap of almost 7 years and 13 versions elapsed. Another example, the project JavaGeom where the primary source Tervana et al. [51], published in 2017 used a version from 2012 when a more recent one from 2014 was available. Quite interestingly, the eCos project, one of the most studied ones as shown in Table 2, has a single concrete version from 2009. Furthermore, the last publication

Table 4: Use of open source projects across activities and subactivities

	Requirements Engineering	Design	Realisation	Testing
Domain Engineering	[9] [46] [5] [6] [16] [18] [31] [51] [38] [25] [54] [52] [40] [37] [29] [20] [15] [3] [10] [23] [47] [41] [42] [14] [11] [4] [50] [34] [49] [8] [48] [35] [36]	[3]	[5] [6] [18] [31] [51] [38] [52] [37] [36] [23] [30] [55] [42] [14] [11] [4] [50] [34] [8] [35]	[46] [39] [45] [21] [1] [22] [26] [29]
Application Engineering	[46] [25] [54] [20] [47] [41] [50]		[6] [31] [55] [50]	[46] [1]

Maintenance and evolution	[5] [16] [51] [15] [3] [23] [55] [42] [34] [35]
Tooling	[6] [16] [18] [31][51] [38] [54] [37] [29] [20] [36] [21] [15] [3] [1] [10] [22] [30] [47] [55] [41] [42] [14] [26] [11] [4] [50] [49] [8] [25]

by Xue et al. [54] that studied it was published in 2017, thus there is a gap of about 7 years.

Mining the information for Table 3 was not an easy task. There were projects with no concrete versions reported. For example, Parrot, Sendmail, Elevator, Cherokee or ArgoUML. Consequently, it was not possible to calculate or estimate their recency in an accurate way. The websites for projects like uClinux and FraSCAti at the time of writing were no longer accessible, fact which rendered impossible to gather their information. Also, some projects such as Fiasco, do not publish release dates of older versions thus we could not verify the information of some of the reported versions.

A third aspect that we wanted to study was the use of feature models across the development subactivities of a software product line. To classify the primary studies, we employ the framework proposed by Pohl et al. [43]. This framework identifies the two standard software product line activities, *Domain Engineering* and *Application Engineering*, and further divides each of them along four subactivities, namely: *Requirements engineering*, *Design*, *Realisation*, and *Testing*. To these combined eight subactivities, we add *Maintenance and evolution* to classify the primary sources which study any of these two topics, and *Tooling* to classify the primary sources that proposed tool support for their work.

Table 4 summarizes our findings for this aspect. For the *Domain Engineering* activity, its subactivities respectively gathered: 33, 1, 20, and 8 primary sources. For the *Application Engineering* activity, its subactivities respectively obtained: 7, nil, 4, and 2 primary sources. For *Maintenance and evolution* we found 10 primary sources, whereas for *Tooling* we identified 30 primary sources. Not surprisingly, the most frequent uses of feature models were for the subactivities of *Requirements engineering* at both levels. Among the favored themes were reverse engineering feature models and configuration optimization. In second place was tooling, because the majority of primary sources implemented and contributed algorithms in tools necessary for obtaining the reported results. In third place were the primary studies on evolution, e.g. of Linux models, and maintenance, e.g. fixing inconsistencies. An interesting finding was that in 8 primary sources, the main use of feature models was for testing purposes. At the other extreme though, design subactivities gather the fewest number of primary sources.

A fourth aspect of our study was the replication of the results presented in the primary sources. To assess this aspect, we fetched the references to URLs, repositories, and tools presented in the articles for which we checked their availability. We classified the primary sources in four categories: *i)* full replication, meaning that all the data, sources and artifacts deemed necessary for replication were available, *ii)* partial replication, whereby any of these elements were not directly available from the links provided in the articles, *iii)* none, in the case there was no indication of any availability of the required artifacts, and *iv)* not possible to determine when, for instance, authors provided a link that was not accessible at the time of our study. Table 5 summarizes our findings for this aspect. Slightly more than half of the primary sources provide full replication, followed by ten primary sources that provide no resources for replication, seven for which we were not able to ascertain the availability, and two with a partial availability.

The fifth aspect of our study was the format used to represent the feature models. For this aspect, we recorded the names as reported in the primary sources, even though they might be interchangeable (e.g. DIMACS and CNF) or transformable form one to another (e.g. KConfig and DIMACS). We found a total of 15 formats reported with KConfig and DIMACS being the two most frequent ones with 12 primary sources. A far second place was CDL with 5 primary sources, followed closely by FAMA with 4 primary sources. FAMILIAR, SXFM and an adhoc metamodel (this latter for the ArgoUML project) each obtained 2 primary sources. The remaining 8 formats had each one a primary source.

Table 5: Availability for study replication

Level	No	**Primary Sources**
Full	22	[46] [6] [16] [18] [31] [51] [38] [25] [39] [54] [52] [29] [20] [36] [1] [10] [30] [41] [26] [11] [50] [49]
Partial	2	[23] [35]
None	10	[9] [5] [45] [22] [47] [55] [42] [14] [4] [34]
Not possible to determine	7	[40] [37] [21] [15] [3] [8] [48]

Table 6: Model formats used in open source projects

Format	No	Primary Sources
KConfig	12	[9] [5] [16] [37] [36] [15] [1] [10] [8] [48] [35] [23]
DIMACS	12	[18] [54] [29] [20] [21] [22] [30] [47] [26] [4] [49] [23]
CDL	5	[9] [5] [37] [36] [10]
FAMA	4	[38] [39] [45] [46]
FAMILIAR	2	[3] [41]
SXFM	2	[25] [46]
Adhoc	2	[6] [31]
Metamodel		
CNF	1	[51]
BDD	1	[40]
XML	1	[42]
Prolog	1	[14]
FDL	1	[14]
CIDE	1	[11]
SPLConqueror	1	[50]
EvoFM	1	[42]

2.3 Analysis

In this subsection we concisely analyze the most salient findings in our study and the potential avenues for further research.

Massive predominance of Linux and KConfig. Without a shadow of a doubt, the Linux project and its configuration language KConfig dwarf the research efforts performed on any other open source project, in terms of the number of versions studied, their recency, and the number of primary sources that rely on them. We argue that this predominance is not only due to the great impact that this project has had in the computing industry in general, or its extensive use of pre-processor code. Instead, we posit that is fundamentally due to the tool ecosystem that has been built around it (e.g. Undertaker, LVAT or KernelHaven). This ecosystem not only gives access to data that researchers could use for multiple purposes (e.g. testing or extraction of feature models), but also enables researchers to share, contrast, and improve their work (e.g. [17]). We adduce that the replication of this experience for other open source projects can lead to a healthy project diversity, beneficial not only for variability modeling research but for the SPL community at large.

Aging and outdated datasets. Our recency analysis showed that except for Linux, all the datasets coming for the rest of the projects are not being updated at a rate that prevents them from aging. Granted, the nature of the research in SPL or the dynamic nature of several open source communities may impede an *in vivo* analysis, but nonetheless, we argue that a broad gap between the current project versions and those analyzed, represents a significant threat to validity to any empirical study. We hope that the awareness and acknowledgment of this fact sparks in the community interest in developing other tools and projects ecosystems to overcome and prevent this expanding gap.

Importance of artifacts beyond feature models and source code. It is not a surprise that the use of feature models pervades the entire spectrum of SPL subactivities. Naturally, open source projects have been studied across almost all these subactivities. However, the great majority of primary sources concentrate on source code as the sole artifact mined from repositories. This choice effectively hampers any further benefits that could be obtained from open source projects. Enriching the variety of artifacts mined can certainly benefit researchers who work on specific subactivities of SPLs like testing. A salient example of an enriched data set is the work by Abal et al. who mined variability bugs to create a database where bugs are reported along with their fixes [2].

Replication as a pillar of empirical SPL research. SPL research as well as other areas in software engineering have experienced an increasing adherence to open science and empirical research practices that advocate the open access to data, algorithms, and any artifacts that allows the replication of research studies. In our study, slightly more than half of the primary sources had their resources available for replication. However, availability of resources is not a surrogate of full experimental replication, which based on our experience is severely lacking in SPL research. Adopting and exploiting container technologies such as Docker, as advocated by Cito et al. [13], at a community level can certainly facilitate the distribution of artifacts and data, and enable full study replication.

Building bridges with mainstream tool communities to enrich SPL research and further widen its impact. Our study confirmed the remarkable impact that KConfig has had in SPL research because it is the most studied variability language and tool. However, its influence has been, to the best of our understanding, primarily unidirectional. That is, the results in the realm of SPL research have not fed back into the Linux and KConfig user communities. Hence the impact of this research unfortunately remains contained within the boundaries of the SPL community. But there are other mainstream communities that share many similar interests related to SPL for which it would be worth exploring and building bridges, to and from, that could not only benefit SPL research but also could serve as a vector to spread its results. Candidates could be among the large array of build and continuous integration mainstream tools that have recently taken centre stage, such as Maven, Gradle, Jenkins, Travis, etc.

2.4 Threats to validity

We acknowledge that because of our choice of relying on the 242 articles selected by the study of Galindo et al. (see [19]), we inherit the threats to validity of their work. Thus, different choices of search terms, search engines, cut off dates, ranking and selection criteria, and broader scope beyond feature models might have produced a different set of primary sources for our study. These all are valid angles that we will explore in our future work. Another threat to validity concerns the way the data was extracted for our study. We followed a systematic approach where the selection and collection of data of our primary sources were performed independently by the authors and later discussed until consensus was reached, followed by a thorough and cross-checked synthesis, analysis, and report.

3 RELATED WORK

Ecological validity has always been an important concern for researchers of SPLs. Consequently, there has been a wide breadth

and depth spectrum of industrial research projects. A recent review by Rabiser et al. makes a comparison of research done in industry versus the research carried out in academia, and did not find any fundamental gaps in terms of the topics of study over the last two decades [44]. As another example, work by Knüppel et al. analyze the differences between feature models in the "real world" versus academic ones and propose ways to bridge the differences [28]. However, to the best of our knowledge, except for a few cases that report the provenance of case studies (e.g. [32, 33]), there has not been a systematic analysis on the impact of open source projects in SPL research.

4 CONCLUSIONS AND FUTURE WORK

In this paper we report our first findings on the impact of open source projects as a source to enhance ecological validity in SPL research with a special focus on feature models. In the pool of 41 primary sources of our study, we identified 43 open source projects. Among them, Linux was the project most studied in terms of versions and number of primary sources. We also found that except for Linux, the versions studied of the rest of the projects are not recent. On the contrary, many of them exhibit wide gaps of five years or more with the most recent project version released. In the projects identified, the features models were used in all but one of SPL activities (i.e. design at Application Engineering) and in conjunction with source code artifacts in the great majority of cases. In addition, our work revealed that the slight majority of primary sources had their artifacts available for replication, and that the most used format for representing feature models was KConfig and their derived DIMACS representation.

For our future work, we plan to address the related threats to validity identified by expanding the scope of our study to consider other search terms and search engines, publication dates, breadth of topics beyond feature models and variability modeling, and assess replication beyond artifact availability. Despite the shortcomings documented and the challenges that lay ahead, we hope that our work helps to rekindle the interest in open source projects as an alternative source of and ground for experimentation in variability modeling and other SPL areas.

ACKNOWLEDGMENTS

This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) grant RGPIN-2017-05421.

REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the linux kernel: a qualitative analysis. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 421–432. https://doi.org/10.1145/2642937.2642990

[2] Iago Abal, Jean Melo, Stefan Stanciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wasowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *ACM Trans. Softw. Eng. Methodol.* 26, 3 (2018), 10:1–10:34. https://doi.org/10.1145/3149119

[3] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. 2014. Extraction and evolution of architectural variability models in plugin-based systems. *Software and System Modeling* 13, 4 (2014), 1367–1394. https://doi.org/10.1007/s10270-013-0364-2

[4] Nele Andersen, Krzysztof Czarnecki, Steven She, and Andrzej Wasowski. 2012. Efficient synthesis of feature models. In *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1*. 106–115. https://doi.org/10.1145/2362536.2362553

[5] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. 2017. Automated Repairing of Variability Models. In *Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume A, Sevilla, Spain, September 25-29, 2017*. 9–18. https://doi.org/10.1145/3106195.3106206

[6] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Multi-objective reverse engineering of variability-safe feature models based on code dependencies of system variants. *Empirical Software Engineering* 22, 4 (2017), 1763–1794. https://doi.org/10.1007/s10664-016-9462-4

[7] 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

[8] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Feature-to-Code Mapping in Two Large Product Lines. In *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*. 498–499. https://doi.org/10.1007/978-3-642-15579-6_48

[9] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2010. Variability modeling in the real: a perspective from the operating systems domain. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto (Eds.). ACM, 73–82. https://doi.org/10.1145/1858996.1859010

[10] 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

[11] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, and Paulo Borba. 2012. Intraprocedural dataflow analysis for software product lines. In *Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD 2012, Potsdam, Germany, March 25-30, 2012*, Robert Hirschfeld, Éric Tanter, Kevin J. Sullivan, and Richard P. Gabriel (Eds.). ACM, 13–24. https://doi.org/10.1145/2162049.2162052

[12] Andrade Chittaranjan. 2018. Internal, External, and Ecological Validity in Research Design, Conduct, and Evaluation. *Indian journal of psychological medicine* 40, 5 (2018), 498–499. https://doi.org/doi:10.4103/IJPSYM.IJPSYM_334_18

[13] Jürgen Cito and Harald C. Gall. 2016. Using docker containers to improve reproducibility in software engineering research. In *Proceedings of the 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, 906–907. https://doi.org/10.1145/2889160.2891057

[14] Robertas Damasevicius, Paulius Paskevicius, Eimutis Karciauskas, and Romas Marcinkevicius. 2012. Automatic Extraction of Features and Generation of Feature Models from Java Programs. *ITC* 41, 4 (2012), 376–384. https://doi.org/10.5755/j01.itc.41.4.1108

[15] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2014. Extracting feature model changes from the Linux kernel using FMDiff. In *The Eighth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '14, Sophia Antipolis, France, January 22-24, 2014*. 22:1–22:8. https://doi.org/10.1145/2556624.2556631

[16] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2017. Analysing the Linux kernel feature model changes using FMDiff. *Software and System Modeling* 16, 1 (2017), 55–76. https://doi.org/10.1007/s10270-015-0472-2

[17] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the Kconfig semantics and its analysis tools. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2015, Pittsburgh, PA, USA, October 26-27, 2015*, Christian Kästner and Aniruddha S. Gokhale (Eds.). ACM, 45–54. https://doi.org/10.1145/2814204.2814222

[18] 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, SPLC 2017, Volume A, Sevilla, Spain, September 25-29, 2017*. 19–28. https://doi.org/10.1145/3106195.3106208

[19] José A. Galindo, David Benavides, Pablo Trinidad, Antonio Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated analysis of feature models: Quo vadis? *Computing* 101, 5 (2019), 387–433. https://doi.org/10.1007/s00607-018-0646-1

[20] 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 *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 517–528. https://doi.org/10.1109/ICSE.2015.69

[21] 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

[22] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. Assessing Software Product Line Testing Via Model-Based Mutation: An Application to Similarity Testing. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*. 188–197. https://doi.org/10.1109/ICSTW.2013.30

[23] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. Towards automated testing and fixing of re-engineered feature models. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 1245–1248. https://doi.org/10.1109/ICSE.2013.6606689

[24] Ruben Heradio, Hector Perez-Morago, David Fernández-Amorós, Francisco Javier Cabrerizo, and Enrique Herrera-Viedma. 2016. A bibliometric analysis of 20 years of research on software product lines. *Information & Software Technology* 72 (2016), 1–15. https://doi.org/10.1016/j.infsof.2015.11.004

[25] Robert M. Hierons, Miqing Li, Xiaohui Liu, Sergio Segura, and Wei Zheng. 2016. SIP: Optimal Product Selection from Feature Models Using Many-Objective Evolutionary Optimization. *ACM Trans. Softw. Eng. Methodol.* 25, 2 (2016), 17:1–17:39. https://doi.org/10.1145/2897760

[26] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An algorithm for generating t-wise covering arrays from large feature models. In *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1*. 46–55. https://doi.org/10.1145/2362536.2362547

[27] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071. https://doi.org/10.1007/s10664-015-9393-5

[28] 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 *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, 291–302. https://doi.org/10.1145/3106237.3106252

[29] Jia Hui (Jimmy) 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, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, Douglas C. Schmidt (Ed.). ACM, 91–100. https://doi.org/10.1145/2791060.2791070

[30] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable analysis of variable software. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 81–91. https://doi.org/10.1145/2491411.2491437

[31] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability extraction and modeling for product variants. *Software and System Modeling* 16, 4 (2017), 1179–1199. https://doi.org/10.1007/s10270-015-0512-y

[32] 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). https://doi.org/10.1002/smr.1912

[33] Roberto E. Lopez-Herrejon, Lukas Linsbauer, and Alexander Egyed. 2015. A systematic mapping study of search-based software engineering for software product lines. *Information & Software Technology* 61 (2015), 33–51. https://doi.org/10.1016/j.infsof.2015.01.008

[34] Roberto E. Lopez-Herrejon, Leticia Montalvillo-Mendizabal, and Alexander Egyed. 2011. From Requirements to Features: An Exploratory Study of Feature-Oriented Refactoring. In *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*, Eduardo Santana de Almeida, Tomoji Kishi, Christa Schwanninger, Isabel John, and Klaus Schmid (Eds.). IEEE Computer Society, 181–190. https://doi.org/10.1109/SPLC.2011.52

[35] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Evolution of the Linux Kernel Variability Model. In *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*. 136–150. https://doi.org/10.1007/978-3-642-15579-6_10

[36] 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 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). 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 Trans. Software Eng.* 41, 8 (2015), 820–841. https://doi.org/10.1109/TSE.2015.2415793

[38] Ganesh Khandu Narwane, José A. Galindo, Shankara Narayanan Krishna, David Benavides, Jean-Vivien Millo, and S. Ramesh. 2016. Traceability Analyses between Features and Assets in Software Product Lines. *Entropy* 18, 8 (2016), 269. https://doi.org/10.3390/e18080269

[39] José Antonio Parejo, Ana B. Sánchez, Sergio Segura, Antonio Ruiz Cortés, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2016. Multi-objective test case prioritization in highly configurable systems: A case study. *Journal of Systems and Software* 122 (2016), 287–310. https://doi.org/10.1016/j.jss.2016.09.045

[40] Hector Perez-Morago, Ruben Heradio, David Fernández-Amorós, Roberto Bean, and Carlos Cerrada. 2015. Efficient Identification of Core and Dead Features in Variability Models. *IEEE Access* 3 (2015), 2333–2340. https://doi.org/10.1109/ACCESS.2015.2498764

[41] Andreas Pleuss and Goetz Botterweck. 2012. Visualization of variability and configuration options. *STTT* 14, 5 (2012), 497–510. https://doi.org/10.1007/s10009-012-0252-z

[42] Andreas Pleuss, Goetz Botterweck, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. 2012. Model-driven support for product line evolution on feature level. *Journal of Systems and Software* 85, 10 (2012), 2261–2274. https://doi.org/10.1016/j.jss.2011.08.008

[43] K. Pohl, G. Bockle, and F. J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[44] 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 *Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018*. 14–24. https://doi.org/10.1145/3233027.3233028

[45] Ana B. Sánchez, Sergio Segura, and Antonio Ruiz Cortés. 2014. The Drupal framework: a case study to evaluate variability testing techniques. In *The Eighth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '14, Sophia Antipolis, France, January 22-24, 2014*. 11:1–11:8. https://doi.org/10.1145/2556624.2556638

[46] 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. https://doi.org/10.1007/s10270-015-0459-z

[47] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany 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 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 465–474. https://doi.org/10.1109/ASE.2013.6693104

[48] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2010. The Variability Model of The Linux Kernel. 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, 45–51. http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf

[49] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2011. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 461–470. https://doi.org/10.1145/1985793.1985856

[50] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don S. Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting performance via automated feature-interaction detection. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 167–177. https://doi.org/10.1109/ICSE.2012.6227196

[51] Xhevahire Tërnava and Philippe Collet. 2017. Early Consistency Checking between Specification and Implementation Variabilities. In *Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume A, Sevilla, Spain, September 25-29, 2017*. 29–38. https://doi.org/10.1145/3106195.3106209

[52] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 178–188. https://doi.org/10.1109/ICSE.2015.39

[53] 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

[54] Yinxing Xue, Jinghui Zhong, Tian Huat Tan, Yang Liu, Wentong Cai, Manman Chen, and Jun Sun. 2016. IBED: Combining IBEA and DE for optimal feature selection in software product line engineering. *Appl. Soft Comput.* 49 (2016), 1215–1231. https://doi.org/10.1016/j.asoc.2016.07.040

[55] 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*. 417–422. https://doi.org/10.1109/CSMR.2012.52

Textual Variability Modeling Languages

An Overview and Considerations

Maurice H. ter Beek
ISTI–CNR, Pisa, Italy
maurice.terbeek@isti.cnr.it

Klaus Schmid
University of Hildesheim, Germany
schmid@sse.uni-hildesheim.de

Holger Eichelberger
University of Hildesheim, Germany
eichelberger@sse.uni-hildesheim.de

ABSTRACT

During the three decades since the invention of the first variability modeling approach [28], there have been multiple attempts to introduce advanced variability modeling capabilities. More recently, we have seen increased attention on textual variability modeling languages. In this paper, we summarize the main capabilities of state of the art textual variability modeling languages, based on [23], including updates regarding more recent work. Based on this integrated characterization, we provide a discussion of additional concerns, opportunities and challenges that are relevant for designing future (textual) variability modeling languages. The paper also summarizes relevant contributions by the authors as input to further discussions on future (textual) variability modeling languages.

CCS CONCEPTS

• **Software and its engineering → Specification languages**; **Software product lines**.

KEYWORDS

software product lines, variability modeling, textual specification languages

ACM Reference Format:
Maurice H. ter Beek, Klaus Schmid, and Holger Eichelberger. 2019. Textual Variability Modeling Languages: An Overview and Considerations. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3307630.3342398

1 INTRODUCTION

Since the very early days, variability modeling has mostly focused on graphical modeling [28], especially using feature diagrams in the form of trees. This has led to different notations, which often only varied in minor technical details [41]. Variability modeling was handled in numerous ways in practice using textual notations. Classical examples of these are KConfig [29] and CDL [43], textual variability description languages that have been created in the open source world. However, these kinds of languages typically suffer from the problem that they are not formally defined and very hard to analyze [25].

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342398

Textual variability modeling approaches were also invented in academia. Most follow the notion of feature modeling [41] with variations. Some also took the approach of decision modeling [40].

The goal of this paper is to summarize and update the categorization of textual variability modeling languages provided in [23] as a basis for discussing future options and challenges towards the design of a simple (textual) variability modeling language that the community can agree on. Due to space restrictions, we can, of course, not replicate that earlier survey. Thus, we also refer the reader to this earlier publication for further details [23].

2 A SHORT STATE OF THE ART OF TEXTUAL VARIABILITY MODELING LANGUAGES

In this section, we provide a summary of the overview on textual variability modeling languages presented in [23], updated with more recent approaches. This comprehensive review of existing textual variability modeling languages was based on both an analysis of the existing literature as well as contact with the corresponding authors to include potential feedback in order to ensure that the categorizations of the various languages was adequate. Thus, there may be (implemented) language capabilities that are not mentioned in the underlying literature or indicated by the involved authors and, therefore, not listed in this paper.

2.1 Updated Literature Analysis

The original analysis (cf. [23, Sect. 4]) considered these languages:

- *Feature Description Language (FDL)* [42] mainly aims at being a textual representation of feature diagrams.
- *Forfamel* [5] is part of the Kumbang approach; Forfamel aims at feature modeling, while Koalish adds structural modeling.
- *Tree grammars for representing cardinality-based feature models were introduced by Batory in [6].*
- *Variability Specification Language (VSL)* [1] integrates feature modeling with configuration links and variable entities.
- *Simple XML Feature Model (SXFM)* [32] is an XML-based representation of feature models.
- *FAMILIAR* [2], next to modeling variability, also includes capabilities for combining and analyzing variability models.
- *Text-based Variability Language (TVL)* [17] supports textual feature modeling, including capabilities for feature attributes, cardinalities and modularization.
- *μTVL* [16] is a variation of TVL, dropping some concepts, but also adding others like multiple trees in a single model.
- *CLAss, FEature, Reference approach (Clafer)* [13] combines meta-modeling of classes with feature modeling support.
- *VELVET* [35] is a language, inspired by TVL, but extends it in several directions and reimplements it from scratch.

- *INDENICA Variability Modeling Language (IVML)* [39] follows the decision modeling paradigm with a strong focus on ease of learnability, expressiveness, and scalability.

In the meantime, some additional approaches have been published:

- *Clafer (extended with behavior)* [27] extends [13] with a temporal dimension, resulting in a language combining behavior, structure, and variability.
- *PyFML* [3] is a textual feature modeling language based on the Python programming language.
- *Variability Modeling (VM)* [4] is a language that was developed in an industrial project, with specific constraints to ease reasoning particularly for applications in the video domain.

Although we are aware of more recent XML-based approaches than SXFM, such as, e.g., [45], we do not include them here, in particular if they do not add further capabilities or aim at a pure XML representation (on instance or schema level) of existing capabilities. The main focus of [23] was on classifying and summarizing the characteristics that are actually supported by the languages to better understand their peculiarities. Discussions of potential future capabilities as well as secondary characteristics like analyzability were excluded. We will discuss these aspects in Section 3. Thus, the subset of characteristics we consider in this section as a basis for our discussion in this paper is based on the following dimensions:

- Configurable elements
- Constraint support
- Configuration support
- Scalability support
- Language characteristics

In Table 1, we show a selection of the extensive classification of textual variability modeling languages from [23], updated for Clafer (extended with behavior), PyFML, and VM. From the above dimensions, we focus here on particular sub-dimensions (stated below between parentheses): configurable elements (forms of variation, attached information, cardinalities, references, and additional data types — such as basic types predefined by the language, user-defined types, or types derived from already known types), constraint support (constraint expressions), configuration support (default values, value assignment, and partial or complete configurations), scalability support (here only through composition, i.e., the capability of integrating units of configurable elements into a single model). Finally, we briefly discuss language characteristics, as reported in Table 7 in [23], in Section 3. As an additional dimension, we also report whether the language has a formal semantics.

We refer the interested reader to [23] for detailed pointers (including page numbers) to the literature that confirm the level of support offered by the surveyed textual variability languages. In Table 1, we simplify the notation to direct (+), indirect (±), unclear (?) or no (−) support. In the next sections, we summarize the classification, providing information on the type of attached information, cardinalities, and references that is supported, as well as details of the supported data types and to what they apply.

Compared with [23], we note the following updates. Clafer (extended with behavior) provides direct support for *simple* cross-tree constraints (cf. [27, p7]) and for so-called parallel decomposition of non-exclusive clafers, i.e., the selection of *multiple* features out of several possible variations (cf. [27, p13]).

PyFML provides direct support for *optional*, *alternative*, and *multiple* feature selection, for *attached information* in the form of attributes of *predefined* types (Boolean, Integer, Float, String), *simple* cross-tree constraints, constraint expressions in *propositional* logic as well as both *relational* and *arithmetic* constraint expressions, and configuration support by allowing *default values* and *value assignment* [3, p46].

As can be concluded from [4, Sect. 5], VM provides similar capabilities as PyFML, but also supports the specification of *cardinalities*, a modularization mechanism supporting *composition*, direct support for *partial configurations* and indirect support for *complete configurations*. In addition (not detailed in Table 1), VM supports constraint resolution hints such as delta values or objective functions.

From [23], we know that TVL and μTVL do not provide direct support for full-fledged *composition*, but merely for inclusion and conjunction. Finally, we are aware of *formal semantics* for FDL [42, p4ff], Forfamel (indirectly, by translation to WCRL) [5, p36], TVL [17, p1136ff], μTVL [16, p208ff], and Clafer [27, p2:23ff].

2.2 Configurable Elements

The basic elements of configuration in nearly all languages is a *feature*, respectively a *feature group*, the only exceptions are IVML, which uses decision variables, and Clafer, where it is a conceptual mix of structural and variability modeling, called a *clafer*.

All languages support *optional* and *alternative* variability, most do also support *multiple* selection, i.e., selecting at least two out of a range of possible features.

Most languages also cater for *attached information* to the basic variability unit. Mostly these are feature *attributes*. Sometimes this can also be *parameters* (VSL) or other *features* (Clafer). IVML also supports *meta-attributes*, which can, for example, express binding times, implementation advices, etc.

Most languages also support *cardinalities*. Typically, these are *feature* and *group cardinalities*. However, Tree grammars only support *feature cardinalities*, while SXFM and μTVL only support *group cardinalities*. Depending on the language design, *cardinalities* are typically realized as a special capability. In some cases, however, the language supports generic multiplicity and the details of *cardinalities* are expressed as constraints.

Many languages do also support the notion of (configuration) *references*, which simply alias other configurable elements. As most languages are very restricted with respect to what a configurable element can be, typically *references* are only possible to a single type. However, IVML differs insofar as it supports *references* to arbitrary *data types*. This enables much richer static (type) checking. As a consequence, IVML, as shown in [22], can also be used to model topological variability. Topological variability targets variations of connecting components with respect to a certain order, in specific interconnected hierarchies, in different quantities [11], or, in general, in terms of graph-like structures.

Related to the question of the basic elements for expressing variability is the question to what extent *data types* are supported. In most cases, there is a basic *feature* (or *clafer*) data type. The only exception is IVML, which can combine the basic element of variation with all available *data types*.

Table 1: Language support for configurable elements, type systems, constraints, configurations, scalability, and semantics

	forms of variation							data types			constraint expressions					configurations					
Language	optional	alternative	multiple	extension	attached info	cardinalities	references	predefined	derived	user-defined	simple	propositional	first-order	relational	arithmetic	default values	assign values	partial	complete	composition	formal semantics
FDL	+	+	+	−	−	±	−	−	−	−	+	−	−	−	−	+	−	−	−	−	+
Forfamel	+	+	+	+	+	+	+	−	−	+	−	+	+	+	+	−	+	−	+	−	±
Tree grammars	+	+	+	−	−	+	−	−	−	−	−	+	−	−	−	−	−	−	−	−	−
VSL	+	+	+	+	+	+	+	+	−	+	+	?	?	−	?	+	+	+	+	+	−
SXFM	+	+	+	−	−	+	−	−	−	−	−	?	−	−	−	−	−	−	−	−	−
FAMILIAR	+	+	+	−	−	−	?	+	+	−	−	+	−	−	−	−	+	+	+	+	−
TVL	+	+	+	?	+	+	+	+	−	+	−	+	−	+	+	−	+	−	−	−	+
μTVL	+	+	±	+	+	+	−	+	−	−	+	+	−	+	+	−	?	+	+	−	+
Clafer	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	−	+	+	+	+	+
VELVET	±	+	+	+	+	±	+	+	−	−	−	+	−	+	−	+	+	+	±	+	−
IVML	+	+	+	+	+	±	+	+	+	+	−	+	+	+	+	+	+	+	±	+	−
PyFML	+	+	+	−	+	−	−	+	−	−	+	+	−	+	+	+	+	−	−	−	−
VM	+	+	+	−	+	+	−	+	−	−	+	+	−	+	+	+	+	+	±	+	−

Some languages allow the derivation of new types, e.g., through a form of inheritance like in object-oriented languages, type composition, container types, or even composing types with constraints, leading to *type restrictions*. However, in most languages these further *data types* are restricted to feature attributes.

2.3 Constraint Support

The various languages differ considerably in terms of their capabilities for *constraint expressions*. Overall, several layers of expressiveness can be distinguished. As a general rule, these are ordered in increasing levels of expressiveness. However, this correlates also to a decreasing level of analyzability (cf. Section 3).

Especially in diagrammatic presentations often basic requires and excludes relationships are present *(simple dependencies)*. However, in textual languages these are rarely to be found. Rather all languages, except for FDL, at least support full *propositional logic*.

As *simple dependencies* can be seen as special cases of propositional logic, there is no need to have both. However, four languages combine *simple dependencies* and *propositional logic*, probably using *simple dependencies* as shortcuts for otherwise more complex *propositional logic* or just to be close to classical feature modeling publications. *Propositional logic* can be extended, e.g., by supporting *relational expressions* or *arithmetic expressions*.

Some languages also support quantification over formulas, which enables for example to give constraints over all subtrees. In [23], this is (not fully correctly) called *first-order logic*.[1] While this is a very powerful construct, it is only available in four languages considered in this survey: Forfamel, VSL, Clafer, and IVML.

[1]Most languages support the quantifiers, but not necessarily the predicates, functions and constants typical of first-order logic.

A mechanism, which is only available in IVML, is the use of default constraints. These are constraints that can be altered as part of the constraint-resolution process. In particular, together with scoped imports (cf. Section 2.5) this leads to (restricted) support for non-monotonic reasoning.

2.4 Configuration Support

While graphical variability modeling notations are typically focused only on the modeling, it is actually rather common for textual variability modeling languages to support the *configuration* as well. The most elementary category is the *value assignment*, which is supported by all languages but FDL, Tree grammars, and SXFM.

Some languages (e.g., FDL, VSL, VELVET, IVML, PyFML, and VM) also support *default values*, i.e., values that can be overridden at a later stage. In particular, VM differentiates between static and runtime configurations, using runtime tags as annotation to indicate the *binding time* of features and attributes, allowing the code to increase or decrease values at runtime.

Besides these basic capabilities of setting values, many languages also support the notion of a *configuration* as a first-class concept. In particular all languages except FDL, Forfamel, SXFM, and PyFML. These languages allow to designate a range of *value assignments* as a configuration and manage it separately. In most cases this can also be a *partial configuration*. However, not all approaches do support full separation in the sense that arbitrary many configurations can be separately managed from the basic model description.

2.5 Scalability Support

The earliest languages, namely FDL, Forfamel, Tree grammars, and SXFM, do not provide mechanisms for large-scale variability

modeling through *composition*, and neither does one of the most recent ones, PyFML. TVL and μTVL only allow the *inclusion* and *conjunction*, respectively, of models.

The remaining languages all support some form of scalability through *composition*. Clafer and VELVET do so by *inheritence*, whereas FAMILIAR provides two explicit *composition* operators, the 'merge' operator for overlapping and the 'aggregate' operator for disjoint models. VM supports the *import* of so-called (model) packages. IVML supports the *scoped import* of models, which besides the provisioning of namespaces also provides scopes for the reasoning process. It also provides an interface concept.

2.6 Language Characteristics

While fundamentally all languages described here are textual, they conceptually differ significantly. This is related to where they originate from and to the major sources of inspiration they rely on.

Some, especially the early languages, followed the idea of a tree-like feature diagram rather faithfully and focused on providing a corresponding syntax. Other languages used programming languages like C and Java (VSL, TVL, and IVML) or Python (PyFML) as an inspiration for their approach to syntax. There have also been proposals that rely on XML (e.g., SXFM) and languages like OCL (for IVML) and Alloy have been sources of inspirations, too. Finally, VELVET and μTVL are special cases as they themselves rely on another variability language TVL (which is inspired by C). One of the rationales for using programming languages as a basis is to make it easier and more natural for users to apply these languages.

As predominant structures of the languages we basically find only three alternatives. Some are tree-based, i.e., a tree structure is replicated textually, while others are graph-based ones, i.e., focusing on representing textually a graph structure. Finally, some languages are driven by (potentially nested) declarations of variables, which can then be used along with value assignments to represent, for example, tree as well as graph structures.

3 OTHER CONCERNS IN LANGUAGE DESIGN

In this section, we discuss further aspects that we believe to be relevant for making good choices in language design for future (textual) variability modeling languages. We focus here on seven topics, namely first citizen concepts, quantitative variability modeling, ecosystem support, 'exotic' modeling capabilities, binding time, analyzability, and modular language design.

3.1 First Citizen Concept

Almost all the approaches discussed in this paper use some form of feature as their main language concept. Notable exceptions are Clafer, which is based on an amalgamation of classes and features, and IVML, which represents variability decisions in terms of typed variables, i.e., follows the decision modeling paradigm [40]. However, one should also take into account that there strong relationships among the different paradigms, not only on a conceptual level [18], but in some cases even a formal correspondence has been shown [24].

In fact, features in their different notions are still considered as the predominant variability modeling approach in both industry [12] and academia [34]. However, according to [34], several aspects largely remain unexplored, e.g., non-functional (quantitative) properties, or merely exist as research topics that are currently not sufficiently taken up by industry, e.g., software ecosystems, multi-product lines, or dynamic software product lines.

This is in particular an issue, as without industrial cases, more recent topics remain academic ideas not really contributing to the evolution of variability modeling approaches. Moreover, approach and tool qualities like usability or scalability are typically only illustrated in terms of examples or not studied at all [34]. The authors believe that it is time to consider, explore, evaluate, and experiment with alternative first-level modeling concepts that integrate beneficial aspects of feature-based approaches, e.g., hierarchy and decomposition, with perceived advantages of textual variability modeling, e.g., scalability, as well as currently less explored needs of actual and future real-world variability modeling.

3.2 Quantitative Variability Modeling

Recently, there is growing interest in variability modeling (and analysis) techniques that explicitly consider quantitative aspects, which are particularly relevant to non-functional requirements, such as dependability, energy consumption, security, and cost. Since today's software is often embedded in smart and critical systems that run in environments where events affecting the system occur randomly, quantitative variability modeling is currently a hot topic.

This is reflected by the recent panel at VaMoS'19, which addressed questions like "How to incorporate quantities in (textual) languages for variability modeling?" [7], and the forthcoming special issue on quantitative variability modeling and analysis [8]. In [9], a rich, high-level textual DSL for configurable software-intensive systems was defined, with variability defined in terms of features and offering advanced quantitative constraint modeling options. The approach comes with tool support [10] and it can cope with the complexity of (re)configurable systems stemming from variability, behavior, and randomness. A related approach is provided by IVML [23].

3.3 Ecosystem Support

Some languages claim explicit modeling support for variations in software ecosystems. Bosch postulates that ecosystems are a natural extension of classical product lines [14] insofar as extending the notion of variability to open systems. Indeed, analysis of existing software ecosystems like Eclipse or the Linux package system show that open forms of variability description have evolved independently to support both open, distributed development as well as variability management [36]. In contrast to traditional (closed) variability modeling, open variability allows for the extension of the configuration space by variabilities that are not part of the core product line. In particular, an extension needs to be possible to 3rd parties who are not able to change the initial model. Thus, open variability also requires distributed modeling. Further, this requires particular capabilities of variability languages, as discussed in [37], like modularization and hiding of variabilities. While most other variability modeling approaches restrict themselves to 'closed' variability, IVML aims to support also open ecosystems.

Besides the aforementioned requirements, IVML supports defaults and non-monotonic reasoning capabilities, which have been

derived as being necessary from industrial cases in [15]. EASy-Producer, of which IVML is one component, also aims to support concepts like the feature pack approach, which aims to modularize feature groups along with their implementation in software ecosystems [30].

3.4 'Exotic' Modeling Capabilities

Exploring actual and future needs, besides those already discussed above, may require capabilities, which are currently not (well) supported by (textual) variability management approaches. Some examples are topological configurations with related constraints [11], behavioral aspects [31], or the specification of configuration optimization goals [4, 13]. Some proposals were made, e.g., how to model topological variability including constraints and reasoning in an integrated manner [22], or how to specify behavioral aspects through temporal constraints [27]. However, we perceive a certain reservation of the community against such 'exotic' capabilities, which is in contrast to indications of respective practical and industrial needs, such as in [11].

3.5 Binding Time

An important concern in software product line engineering is not only the specific configuration that is defined, but also when the configuration is determined, respectively applied to the system. One should note that these are ultimately two different notions. For example, if a configuration tool is needed for determining the configuration, then this is typically happening during the development process. On the other hand, the value may only have an effect very late in the process, e.g., a configuration may be instantiated when the system starts, a configuration file is read, and programmatic binding (e.g., through class-loading) happens.

Often both notions are referred to by the term *binding time*, although they are notably different. We propose the terms *definition time* and *binding time* in order to differentiate between them. Thus, the term definition time could be used to refer to something like the stages introduced by Czarnecki et al. [19]. Regarding binding time, Dolstra et al. [20] point out that this is not necessarily well-defined as there may be multiple points in time when an instantiation may happen, even for the same variability. They call this timeline variability. We can regard both definition time and binding time as descriptive information about a variability, which can be variable itself. Beyond those two, other aspects like different implementation technologies and so forth may be relevant, too. Hence, these may need to be represented as well. In [38], the term *meta-variability* was proposed for this more generic concept. Along with this, the authors proposed a very generic implementation approach using aspect-oriented programming for timeline variability.

Existing textual variability languages typically do not address the notion of binding time — and even those which do would hardly be able to represent the richness of concepts outlined above, because they restrict themselves to a single category with predefined unique values. They are thus not able to represent both definition time and binding time and are also not able to capture multiple alternatives which are needed for representing timeline variability. Notable exceptions are VM, as anticipated in Section 2.4, and in particular IVML. IVML directly implements the notion of meta-variability by allowing to attach arbitrarily many meta-decisions to any variability, which can also be set-typed to support timeline variability. As both values and meta-variabilities are user-definable, a context-specific binding time granularity as well as issues like technology variabilities can be supported on a per-need basis.

3.6 Analyzability

This is often a major concern as static analysis of variability models and product lines as a whole can provide very valuable assistance both during modeling and in derivation of product lines [44]. Unfortunately, there is an important trade-off between analyzability and expressiveness, as the more expressive a variability language is, the harder it is to analyze it. While propositional language is decidable and very efficient provers exist to handle it, analysis of higher levels of logic is not only significantly less efficient, it may often even be undecidable. On the other hand, the expressiveness of a higher-level logic may make certain things much easier to express, if not be a precondition for being able to represent the situation adequately in the first place.

Based on this observation some of the authors categorized different levels of expressiveness vs. analyzability trade-offs in earlier work in order to create a map of the situation [21], where four main classes could be identified: *basic variability modeling*, basically corresponding to classic feature models and mappable to propositional logic. *Cardinality-based variability modeling* gets particularly complex if potentially unbounded cardinalities are allowed. This leaves then the realm of decidability. Then *non-Boolean variability modeling* brings its own problems in terms of analyzability, e.g., the need for supporting arithmetic theories. Finally, *configuration references* can lead to significant challenges as they allow for arbitrary aliasing. However, this can also be used to great benefit, e.g., in the context of topological modeling [22].

Due to the inherent trade-off involved, any decision regarding the expressiveness of a language should be made carefully. The identification of different classes of expresssiveness also encourages the definition of different language levels within a modular language design as we will discuss below. Each of these could then have different reasoning support. However, the situation is not as simple as it may seem. As has been discussed by Eichelberger et al. [21, Table 1], many other aspects, like whether quantifiers are supported in constraints, also have a significant impact on analyzability and expressiveness. Making all of these aspects simultaneously customizable may create significant complexity.

3.7 Extensible Language Design

Most languages discussed here focus on a complete solution for variability modeling, while the contribution sometimes concentrates on differences, improvements, or additions over an existing language. For example, μTVL and VELVET extend TVL, Clafer was extended by temporal constraints, and VM provides domain-specific improvements over FAMILIAR. Such extensions do not always require a completely new language. One alternative could be an extensible language design, e.g., capabilities of extending a given language or embedding concepts into a host language.

Similar concepts are known from domain-specific languages (DSLs) [26], where external DSLs are complete languages for a certain purpose, while internal or embedded DSLs utilize and extend the concepts of a host language. As many textual variability languages are realized in terms of DSLs using related tooling, designing a variability modeling language for extensibility may support the experimentation and development of new approaches based on existing concepts and implementations.

3.8 Modular Language Design

In addition to the idea of extending a variability modeling language, conceptual differences may also be realized in terms of different *language levels*. In addition to pure language management (and product lines of variability modeling languages), this would allow to explicitly face trade-offs in the language design rather than aiming for a single general-purpose variability modeling approach.

Similar situations exist if languages explicitly combine basic and advanced modeling concepts (e.g., IVML and VM) or provide increasing capabilities of the same language concept, e.g., constraint capabilities (cf. Forfamel, Clafer, or IVML, as indicated in Table 1). In particular, for constraints and some specific modeling concepts, various trade-offs between expressiveness and analyzability do exist, as discussed in [21]. While simple forms of constraints such as pure Boolean expressions can be efficiently analyzed and solved [33], more complex constraints such as quantorized or temporal constraints are not decidable anymore.

In such settings, we can imagine that a *modular language design* could enable the product line engineer to focus on the most appropriate language level(s) for the situation at hand. Moreover, using just the needed language modules may allow for an automated selection of the most appropriate reasoning or analysis mechanisms, in turn leading to better performance or supporting analysis capabilities that are not available for the full language. Ultimately, a modular language design can foster the reuse of language levels (including the underlying language infrastructure), support the development of domain specific modeling capabilities (as suggested in [4]), and ease experiments as well as prototyping and the development of new variability modeling concepts.

4 CONCLUSION

We have presented an overview of the main characteristics of thirteen textual variability modeling languages, based on a systematic literature analysis reported in [23]. Beyond the coverage of this survey, we have incorporated two recently introduced languages (PyFML and VM) and updated the knowledge about an extended language (Clafer with behavior). Further, we have focused on the following five dimensions: support for configurable elements, including type systems, constraints, configuration, scalability, and formal semantics. Table 1 summarizes these results.

Together with [23], our overview provides an important resource for researchers as well as practitioners in the field of systems and software product line engineering on currently available textual variability modeling languages.

Given the large number of existing textual variability modeling languages, an obvious question is whether we need more languages. Our answer to this is as follows:

We do not just need further (textual) languages for existing concepts, leading to an even greater plethora of variability modeling languages, rather we need more innovative (textual) variability modeling approaches. As an indication for future directions, we discussed additional concerns in language design, including the choice of the first citizen concept, quantitative variability modeling, binding times, ecosystem support, exotic capabilities, and analyzability. In particular, we believe that future languages should have an extensible and modular language design with sub-languages catering for different needs, but holistically integrated into an over-arching concept.

ACKNOWLEDGEMENTS

This work is partially supported by the ITEA3 project REVaMP^2, 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.

We thank the anonymous reviewers for their comments and suggestions that helped us improve the paper.

REFERENCES

[1] Andreas Abele, Yiannis Papadopoulos, David Servat, Martin Törngren, and Matthias Weber. 2010. The CVM Framework – A Prototype Tool for Compositional Variability Management. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10) (ICB Research Report)*, David Benavides, Don S. Batory, and Paul Grünbacher (Eds.), Vol. 37. Universität Duisburg-Essen, 101–105.

[2] 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. https://doi.org/10.1016/j.scico.2012.12.004

[3] Ali Fouad Al-Azzawi. 2018. PyFML – A Textual Language For Feature Modeling. *International Journal of Software Engineering & Applications* 9, 1 (2018), 41–53. https://doi.org/10.5121/ijsea.2018.9104

[4] Mauricio Alférez, Mathieu Acher, José A. Galindo, Benoit Baudry, and David Benavides. 2019. Modeling variability in the video domain: language and experience report. *Software Quality Journal* 27, 1 (2019), 307–347. https://doi.org/10.1007/s11219-017-9400-8

[5] Timo Asikainen, Tomi Männistö, and Timo Soininen. 2006. A Unified Conceptual Foundation for Feature Modelling. In *Proceedings of the 10th International Software Product Line Conference (SPLC'06)*. IEEE, 31–40. https://doi.org/10.1109/SPLINE.2006.1691575

[6] Don S. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Software Product Lines Conference (SPLC'05) (LNCS)*, Henk Obbink and Klaus Pohl (Eds.), Vol. 3714. Springer, 7–20. https://doi.org/10.1007/11554844_3

[7] 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, 13:1–13:2. https://doi.org/10.1145/3302333.3302349

[8] Maurice H. ter Beek and Axel Legay. 2019. Quantitative Variability Modeling and Analysis. *International Journal on Software Tools for Technology Transfer* (2019).

[9] Maurice H. ter Beek, Axel Legay, Alberto Lluch Lafuente, and Andrea Vandin. 2018. A framework for quantitative modeling and analysis of highly (re)configurable systems. *IEEE Transactions in Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2853726

[10] Maurice H. ter Beek, Axel Legay, Alberto Lluch Lafuente, and Andrea Vandin. 2018. QFLan: A Tool for the Quantitative Analysis of Highly Reconfigurable Systems. In *Proceedings of the 22nd International Symposium on Formal Methods (FM'18) (LNCS)*, Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik de Vink (Eds.), Vol. 10951. Springer, 329–337. https://doi.org/10.1007/978-3-319-95582-7_19

[11] Thorsten Berger, Ştefan Stănciulescu, Ommund Øgård, Øystein Haugen, Bo Larsen, and Andrzej Wąsowski. 2014. To Connect or Not to Connect: Experiences from Modeling Topological Variability. In *Proceedings of the 18th International Software Product Line Conference (SPLC'14)*. ACM, 330–339. https://doi.org/10.1145/2648511.2648549

[12] 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. https://doi.org/10.1109/TSE.2013.34

[13] Kacper Bąk, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled. In *Proceedings of the 3rd International Conference on Software Language Engineering (SLE'10) (LNCS)*, Brian A. Malloy, Steffen Staab, and Mark van den Brand (Eds.), Vol. 6563. Springer, 102–122. https://doi.org/10.1007/978-3-642-19440-5_7

[14] Jan Bosch. 2009. From Software Product Lines to Software Ecosystems. In *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*. Carnegie Mellon University, 111–119.

[15] Hendrik Brummermann, Markus Keunecke, and Klaus Schmid. 2012. Formalizing Distributed Evolution of Variability in Information System Ecosystems. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'12)*. ACM, 11–19. https://doi.org/10.1145/2110147.2110149

[16] Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. 2012. Variability Modelling in the ABS Language. In *Proceedings of the 9th International Symposium on Formal Methods for Components and Objects (FMCO'10) (LNCS)*, Bernhard Aichernig, Frank de Boer, and Marcello Bonsangue (Eds.), Vol. 6957. Springer, 204–224. https://doi.org/10.1007/978-3-642-25271-6_11

[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* 11, 12 (2011), 1130–1143. https://doi.org/10.1016/j.scico.2010.10.005

[18] 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 6th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'12)*. ACM, 173–182. https://doi.org/10.1145/2110147.2110167

[19] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Staged Configuration through Specialization and Multi-Level Configuration of Feature Models. *Software Process Improvement and Practice* 10, 2 (2005), 143–169. https://doi.org/10.1002/spip.225

[20] Eelco Dolstra, Gert Florijn, Merijn de Jonge, and Eelco Visser. 2003. Capturing Timeline Variability with Transparent Configuration Environments. In *ICSE Workshop on Software Variability Management (SVM'03)*, Peter Knauber and Jan Bosch (Eds.). IEEE. https://doi.org/10.1109/ICSE.2003.1201282

[21] Holger Eichelberger, Christian Kröher, and Klaus Schmid. 2013. An Analysis of Variability Modeling Concepts: Expressiveness vs. Analyzability. In *Proceedings of the 13th International Conference on Software Reuse (ICSR'13) (LNCS)*, John Favaro and Maurizio Morisio (Eds.), Vol. 7925. Springer, 32–48. https://doi.org/10.1007/978-3-642-38977-1_3

[22] Holger Eichelberger, Cui Qin, Roman Sizonenko, and Klaus Schmid. 2016. Using IVML to Model the Topology of Big Data Processing Pipelines. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC'16)*. ACM, 204–208. https://doi.org/10.1145/2934466.2934476

[23] Holger Eichelberger and Klaus Schmid. 2015. Mapping the design-space of textual variability modeling languages: a refined analysis. *International Journal on Software Tools for Technology Transfer* 17, 5 (2015), 559–584. https://doi.org/10.1007/s10009-014-0362-x

[24] Sascha El-Sharkawy, Stephan Dederichs, and Klaus Schmid. 2012. From Feature Models to Decision Models and Back Again: An Analysis Based on Formal Transformations. In *Proceedings of the 16th International Software Product Line Conference (SPLC'12)*. ACM, 126–135. https://doi.org/10.1145/2362536.2362555

[25] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the Kconfig Semantics and Its Analysis Tools. In *Proceedings of the 14th International Conference on Generative Programming (GPCE'15)*. ACM, 45–54. https://doi.org/10.1145/2814204.2814222

[26] Martin Fowler. 2010. *Domain Specific Languages*. Addison-Wesley Professional.

[27] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. 2019. Clafer: Lightweight Modeling of Structure, Behaviour, and Variability. *The Art, Science, and Engineering of Programming* 3, 1 (2019), 2:1–2:62. https://doi.org/10.22152/programming-journal.org/2019/3/2

[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 CMU/SEI-90-TR-21. Carnegie Mellon University.

[29] KConfig Language [n.d.]. http://kernel.org/doc/Documentation/kbuild/kconfig-language.txt.

[30] Markus Keunecke, Hendrik Brummermann, and Klaus Schmid. 2013. The Feature Pack Approach: Systematically Managing Implementations in Software Ecosystems. In *Proceedings of the 8th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'14)*. ACM, 20:1–20:7. https://doi.org/10.1145/2556624.2556639

[31] Anna-Lena Lamprecht, Stefan Naujokat, and Ina Schaefer. 2013. Variability Management beyond Feature Models. *IEEE Computer* 46, 11 (2013), 48–54. https://doi.org/10.1109/MC.2013.299

[32] Marcilio Mendonça, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T. – Software Product Lines Online Tools. In *Companion Proceedings of the 24th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09)*. ACM, 761–762. https://doi.org/10.1145/1639950.1640002

[33] 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 (SPLC'09)*. Carnegie Mellon University, 231–240.

[34] 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 Systems and Software Product Line Conference (SPLC'18)*. ACM, 14–24. https://doi.org/10.1145/3233027.3233028

[35] 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, 11–20. https://doi.org/10.1145/1944892.1944894

[36] Klaus Schmid. 2010. Variability Modeling for Distributed Development – A Comparison with established practice. In *Proceedings of the 14th International Conference on Software Product Line Engineering (SPLC'10) (LNCS)*, Jan Bosch and Jaejoon Lee (Eds.), Vol. 6287. Springer, 155–165. https://doi.org/10.1007/978-3-642-15579-6_11

[37] Klaus Schmid. 2013. Variability Support for Variability-Rich Software Ecosystems. In *Proceedings of the 4th International Workshop on Product LinE Approaches in Software Engineering (PLEASE'13)*. IEEE, 5–8. https://doi.org/10.1109/PLEASE.2013.6608654

[38] Klaus Schmid and Holger Eichelberger. 2008. Model-Based Implementation of Meta-Variability Constructs: A Case Study using Aspects. In *Proceedings of the 2nd International Workshop on Variability Modeling of Software-intensive Systems (VAMOS'08) (ICB Research Report)*, Patrick Heymans, Kyo C. Kang, Andreas Metzger, and Klaus Pohl (Eds.), Vol. 22. Universität Duisburg-Essen, 63–71.

[39] Klaus Schmid, Christian Kröher, and Sascha El-Sharkawy. 2018. Variability Modeling with the Integrated Variability Modeling Language (IVML) and EASyproducer. In *Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC'18)*. ACM, 306–306. https://doi.org/10.1145/3233027.3233057

[40] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. 2011. A Comparison of Decision Modeling Approaches in Product Lines. In *Proceedings of the 5th International Workshop on Variability Modeling of Software-intensive Systems (VaMoS'11)*. ACM, 119–126. https://doi.org/10.1145/1944892.1944907

[41] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *Proceedings of the 14th International Requirements Engineering Conference (RE'06)*. IEEE, 139–148. https://doi.org/10.1109/RE.2006.23

[42] Arie van Deursen and Paul Klint. 2002. Domain-Specific Language Design Requires Feature Descriptions. *Journal of computing and information technology* 10, 1 (2002), 1–17. https://doi.org/10.2498/cit.2002.01.01

[43] Bart Veer and John Dallaway. [n.d.]. The eCos Component Writer's Guide. http://ecos.sourceware.org/docs-latest/cdl-guide/cdl-guide.html.

[44] Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm, and Ina Schaefer. 2013. The PLA Model: On the Combination of Product-Line Analyses. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13)*. ACM, 14:1–14:8. https://doi.org/10.1145/2430502.2430522

[45] Jingang Zhou, Dazhe Zhao, Li Xu, and Jiren Liu. 2012. Do We Need Another Textual Language for Feature Modeling? A Preliminary Evaluation on the XML Based Approach. In *Software Engineering Research, Management and Applications 2012*, Roger Lee (Ed.). Studies in Computational Intelligence, Vol. 430. Springer, 97–111. https://doi.org/10.1007/978-3-642-30460-6_7

On Language Levels for Feature Modeling Notations

Thomas Thüm
TU Braunschweig
Brunswick, Germany

Christoph Seidl
IT University
Copenhagen, Denmark

Ina Schaefer
TU Braunschweig
Brunswick, Germany

ABSTRACT

Configuration is a key enabling technology for the engineering of systems and software as wells as physical goods. A selection of configuration options (aka. features) is often enough to automatically generate a product tailored to the needs of a customer. It is common that not all combinations of features are possible in a given domain. Feature modeling is the de-facto standard for specifying features and their valid combinations. However, a pivotal hurdle for practitioners, researchers, and teachers in applying feature modeling is that there are hundreds of tools and languages available. While there have been first attempts to define a standard feature modeling language, they still struggle with finding an appropriate level of expressiveness. If the expressiveness is too high, the language will not be adopted, as it is too much effort to support all language constructs. If the expressiveness is too low, the language will not be adopted, as many interesting domains cannot be modeled in such a language. Towards a standard feature modeling notation, we propose the use of language levels with different expressiveness each and discuss criteria to be used to define such language levels. We aim to raise the awareness on the expressiveness and eventually contribute to a standard feature modeling notation.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

product lines, variability modeling, feature model, language design, expressiveness, automated analysis

ACM Reference Format:
Thomas Thüm, Christoph Seidl, and Ina Schaefer. 2019. On Language Levels for Feature Modeling Notations. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3307630.3342404

1 MOTIVATION

In industrial software and systems engineering, there is an increasing awareness that reuse is central for quality and development efficiency [7, 8]. With software product lines, software-intensive systems are derived from reusable assets [44]. Ideally, software products are generated automatically for a given selection of features [3]. However, such configuration is not only used for software or software-intensive systems, but also for arbitrary products following the vision of mass customization [52]. For instance, in an industrial collaboration we modularized financial products, such as loans, into features for improved productivity and consistency compared to a product-by-product development.

While mass customization is applied to various domains, they all have in common that there are constraints among those features, meaning that not all combinations of features are valid. Constraints can have numerous sources [39] and are typically specified by means of feature modeling [4, 26]. Other related, but less prominent approaches to specify constraints are decision models [18, 24] and orthogonal variability models [44]. Since feature models have been introduced in 1990 [26], numerous feature modeling notations have been proposed [14, 25, 47], including graphical notations such as CVL [22] and textual notations such as TVL [12] and KConfig [19]. There are numerous tools for feature modeling available [35], such as FeatureIDE [34], SPLOT [36], FAMA [6], and Betty [49].

The large number of feature modeling notations and languages is a pivotal hurdle for practitioners, researchers, and teachers. Practitioners need interoperability to combine different tools. Researchers need tool support to evaluate their research and reduced effort for tool building. Teachers want to give students hands-on experience with feature modeling, which typically involves teaching a number of notations and tools. Ideally, researchers, practitioners, teachers, and tool builders agree on a single feature modeling language, but this is unlikely to happen, as feature modeling is used for various, very different domains. A language that can express everything required for all domains is too much effort to integrate into tools. In contrast, a core language that is easy to support in tools is most likely not sufficient to express relevant industrial domains.

We aim to tackle this problem by proposing the use of language levels so that tool builders can decide which language levels to support. Those levels are designed to achieve a trade-off between the expressiveness of the modeling language and its applicability to different domains and the capabilities required by a respective tool. Our discussion is inspired by the Java Modeling Language (JML), which is a behavioral interface specification language comprising different language levels [31]. JML is used for many different applications and tools [10], where tool builders can decide which language levels they support [31].

2 MAJOR LEVELS FOR FEATURE MODELING

Our experience with applying feature modeling to industrial applications is that their most important capability is the automated reasoning about their constraints. For instance, in an industrial application where Excel was used to specify about 500 features and 2,000 rules among them, we found that 20% of the features were dead (i.e., were not contained in any valid configuration) and

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342404

about half of the rules were redundant (i.e., their removal would not change the set of valid configurations). In contrast, in another industrial project with about 700 features where FeatureIDE [34] was used, we found a number of redundant constraints but not a single dead feature. A likely reason for the absence of dead features is that FeatureIDE highlights dead features directly in the feature diagram. Even though these are just two cases, it indicates that humans are not able to reason about configuration spaces with hundreds of features and thousands of constraints.

Reasoning about feature models is typically performed by translating the feature model into a logical representation [4, 16, 37] and then encoding analyses as satisfiability problems [5, 21, 55]. Satisfiability problems have the advantage that they can be delegated to efficient off-the-shelf solvers, such as SAT solvers, SMT solvers, CSP solvers, or binary decision diagrams. It is even possible to combine several solvers [49] or to implement dedicated data structures being used in combination with state-of-the-art solvers [30, 49]. The encoding as satisfiability problems also has the advantage that explanations for unsatisfiable problems (e.g., dead features or redundant constraints) can be delegated to algorithms computing minimal unsatisfiable cores [2, 29, 41].

The automated analysis of feature models to detect anomalies, such as dead features or redundant constraints [5, 21], is just one of many applications where tools need to reason about the constraints of the feature model. Besides the analysis of the feature model in isolation, there are numerous different analyses that also incorporate other artifacts, such as source code [55]. Reasoning about feature models is used for parsing [27], dead-code analysis [53], code simplification [59], type checking [54], consistency checking [15], dataflow analyses [32], model checking [13], testing [11] including variability-aware execution [40] and sampling [33, 58], product configuration [46], optimization of non-functional properties [51], and variant-preserving refactoring [20].

The large number of applications that require to reason about constraints indicates the importance of solvers for feature modeling. In order to be able to reduce those problems to dedicated solvers, it is vital that the expressiveness of a feature modeling languages is not larger than that of given solvers. For example, SAT solvers can solve satisfiability problems in propositional logic, whereas SMT solvers support fragments of first-order logic [9]. Most feature modeling notations can easily be translated into propositional logic. However, extended feature models supporting numerical attributes or cardinality-based feature models with feature cardinalities cannot easily be encoded with propositional logic. Hence, there is a trade-off between high expressiveness in the feature modeling language and the number of solvers that are applicable to reason about constraints. Clearly, one could argue that we can do all analyses with SMT solvers only, but our experience is that SMT solvers are magnitudes slower than SAT solvers for many satisfiability problems. Consequently, a reduced expressiveness leads to more solvers being applicable and potentially also to faster solving times.

Based on prior discussions about differing expressiveness of SAT and SMT solvers, we propose two major language levels for feature modeling notations: Level 1 for those notations that can be encoded directly as SAT problem and Level 2 for those that can be encoded as SMT problem, but not as a SAT problem. However, it is not yet clear whether these two levels are sufficient. It requires a community effort to identify (a) solvers that are relevant to reason about feature models and (b) classes of those solvers with the same expressiveness. For instance, it is known that SAT solvers and binary decision diagrams can both be used to decide satisfiability of propositional formulas and both have often been applied to reason about feature models. As a consequence, we do not yet know how many levels are necessary, but at least two levels are required.

3 MINOR LEVELS FOR FEATURE MODELING

In the previous section, we argued for language levels aligned with the expressiveness of state-of-the-art solvers. We refer to those levels as major language levels, as those are likely to have the largest impact on the expressiveness of a feature modeling notation. However, these are clearly not the only requirements for feature modeling languages. This can be illustrated easily by considering the standard input language of SAT solvers, namely DIMACS. DIMACS is a standardized format supported by all SAT solvers, in which a propositional formula is specified in conjunctive normal form, and variables are named $1, 2, \ldots, n$. The success of the SAT community is likely also due to having agreed on this standard format, such that a standard format could have similar effects for feature modeling. However, the DIMACS format is not suitable for feature modeling, as feature models tend to explode in conjunctive normal form [28] and as the hierarchy is crucial for its success.

We argue that even for each major level it is not feasible to agree on a set of language constructs, which gives rise to minor language levels for each major level. While major levels are driven by solving, minor levels are driven by two requirements. First, one should take into account existing languages, as import and export to these languages is vital to the adoption for a new language. Ideally, tools would even use the standard language opposed to their own language in the long run. Second, the design of a language should be influenced by requirements of real-world domains. That is, the language needs to be driven at least to some extent by what is required to model real-world constraints. Nevertheless, it is not useful to consider every language construct of existing languages and every requirement from real-world domains.

The expressiveness of numerous early notations for feature modeling languages has been formalized already in 2007 [47]. This can be a starting point for considerations regarding language constructs for certain levels. If removing a language construct from a language does not reduce its expressiveness, it is considered syntactic sugar. Two levels could only differ in syntactic sugar to reduce the burden of tool developers to support too many language constructs. If two levels are not equally expressive, transformations are still possible between each other but they may result in a homomorphism (i.e., they lose information).

We give an example for a trade-off in terms of expressiveness. Some feature modeling notations allow arbitrary propositional formulas as cross-tree constraints (aka. complex constraints), while others only support requires and excludes constraints between a pair of features (aka. simple constraints) [28]. Having only simple constraints is much easier to handle for tools, but domain modelers may need to create numerous additional abstract features [56] to model real-world constraints [28]. In contrast, with complex constraints it is possible to create constraints which are hard to

translate into conjunctive normal form. For instance, to the best of our knowledge, the Linux kernel feature model cannot be translated into conjunctive normal form without introducing new variables. Tools that need to transform the Linux model into conjunctive normal form use the Tseytin transformation. For Linux, the Tseytin transformation results in about 45,000 new variables for a model with 15,000 variables in the original version [43]. Minor levels could be used to distinguish between notations that support complex constraints and others that only support simple constraints.

4 ORTHOGONAL LEVELS

While minor language levels are concrete instantiations of a major language level, there are also language constructs which do not influence the expressiveness of the modeling notations. Hence, those language constructs are somewhat orthogonal to major and minor language levels and may be combined with any of them. We give two concrete examples for such orthogonal language constructs.

First, a common problem is that feature models tend to grow in size. In our experience, feature models with more than 100 features are hard to understand for domain modelers. Furthermore, any change in the model can potentially introduce anomalies, such as dead features, in any other part of the model as constraints are globally visible. A common method in software engineering to achieve modularity is to apply information hiding. For feature models, feature-model interfaces hide some features and some constraints when feature models are composed with other feature models [48, 57]. Numerous techniques have been proposed to compose feature models [17, 23, 38, 45], which could be combined with feature-model interfaces to achieve modularity and, thus, to support the modeling of large configuration spaces in manageable modules. Even if feature-model interfaces are not available in the language, modular reasoning is feasible by means of slicing [1] and the computation of implicit constraints [2]. Such a modularity is possible at every language level. It is an open question whether modularization concepts should be part of the language that has to be supported by all tools or whether it is an optional and then orthogonal language construct for each language level.

Second, there are several existing feature modeling languages that allow to model the evolution of configuration spaces explicitly. Hyper-feature models [50] allow to refer to features in numerous versions, and even constraints can be defined for certain versions only. Temporal feature models [41] allow to capture every possible evolution scenario explicitly. Both, hyper-feature models and temporal feature models support different aspects of evolving configuration spaces and can be added on top of any feature modeling notation. It is a point for discussion whether this needs to be supported by every tool, as the evolution of feature models may also be tracked by version control instead, and new versions of features can also be supported by newly introduced features. However, in industrial projects, we experienced the need for dedicated languages constructs for evolving configuration spaces. A typical example is that domain modelers want to plan certain changes of the feature model for the future, while they still want to fix wrong constraints in the current version in an integrated manner [42].

5 CONCLUSION AND FUTURE WORK

A standard feature modeling notation can have a great impact on the interoperability of tools. Interoperability may help for industrial adoption by practitioners, researchers to use tools in a plug-and-play manner, and teachers to avoid the introduction of numerous notations. We identified that it is unlikely that a single feature modeling notation will be sufficient for all domains and propose to make use of different language levels. First, major language levels are aligned with the expressiveness of solver classes. Second, minor language levels can be defined inside each major language level to address trade-offs for different purposes. Third, orthogonal language levels can be combined with any previously mentioned major or minor language level (e.g., to facilitate modularity or to model evolving configuration spaces). Which concrete language levels should be considered requires a community effort, which we hope to initiate with this article.

ACKNOWLEDGMENTS

We gratefully acknowledge discussions on language levels for automated analysis of feature models with Maurice H. ter Beek. This work has been partially supported by the German Research Foundation within the project VariantSync (TH 2387/1-1).

REFERENCES

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2011. Slicing Feature Models. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 424–427.
[2] Sofia Ananieva, Matthias Kowal, Thomas Thüm, and Ina Schaefer. 2016. Implicit Constraints in Partial Feature Models. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 18–27.
[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.
[4] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 7–20.
[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), 615–708.
[6] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2007. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. Technical Report 2007-01, Lero, 129–134.
[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 *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 7:1–7:8.
[8] 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 *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 73–82.
[9] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. 2009. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands.
[10] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. 2005. An Overview of JML Tools and Applications. *Int'l J. Software Tools for Technology Transfer (STTT)* 7, 3 (2005), 212–232.
[11] Ivan Do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. 2014. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *J. Information and Software Technology (IST)* 56, 10 (2014), 1183–1199.
[12] Andreas Classen, Quentin Boucher, and Patrick Heymans. 2011. A Text-Based Approach to Feature Modelling: Syntax and Semantics of TVL. *Science of Computer Programming (SCP)* 76, 12 (2011), 1130–1143. Special Issue on Software Evolution, Adaptability and Variability.
[13] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Trans. Software Engineering (TSE)* 39, 8 (2013), 1069–1089.

[14] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 173–182.

[15] Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 211–220.

[16] Krzysztof Czarnecki and Andrzej Wąsowski. 2007. Feature Diagrams and Logics: There and Back Again. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, 23–34.

[17] Deepak Dhungana, Paul Grünbacher, Rick Rabiser, and Thomas Neumayer. 2010. Structuring the Modeling Space and Supporting Evolution in Software Product Line Engineering. *J. Systems and Software (JSS)* 83, 7 (2010), 1108–1122.

[18] Rebecca Duray, Peter T. Ward, Glenn W. Milligan, and William L. Berry. 2000. Approaches to Mass Customization: Configurations and Empirical Validation. *J. Operations Management (JOM)* 18, 6 (2000), 605–625.

[19] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the KConfig Semantics and its Analysis Tools. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 45–54.

[20] Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. 2017. Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. In *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 316–326.

[21] JosÃĪ A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel GutiÃĪrrez-Fernández, and Antonio Ruiz-CortÃĪs. 2019. Automated Analysis of Feature Models: Quo Vadis? *Computing* 101, 5 (May 2019), 387–433. https://doi.org/10.1007/s00607-018-0646-1

[22] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. 2008. Adding Standardized Variability to Domain Specific Languages. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, 139–148.

[23] Gerald Holl, Paul Grünbacher, and Rick Rabiser. 2012. A Systematic Review and an Expert Survey on Capabilities Supporting Multi Product Lines. *J. Information and Software Technology (IST)* 54, 8 (2012), 828–852.

[24] Arnaud Hubaux, Dietmar Jannach, Conrad Drescher, Leonardo Murta, Tomi Männistö, Krzysztof Czarnecki, Patrick Heymans, Tien N. Nguyen, and Markus Zanker. 2012. Unifying Software and Product Configuration: A Research Roadmap. In *Proc. Configuration Workshop (ConfWS)*. 31–35.

[25] Arnaud Hubaux, Thein Than Tun, and Patrick Heymans. 2013. Separation of Concerns in Feature Diagram Languages: A Systematic Survey. *Comput. Surveys* 45, 4, Article 51 (2013), 51:1–51:23 pages.

[26] 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 CMU/SEI-90-TR-21. Software Engineering Institute.

[27] 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 *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 805–824.

[28] 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.

[29] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. 2016. Explaining Anomalies in Feature Models. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 132–143.

[30] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. 2018. Propagating Configuration Decisions with Modal Implication Graphs. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 898–909.

[31] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. 2013. *JML Reference Manual*.

[32] 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.

[33] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 643–654.

[34] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.

[35] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, and Gunter Saake. 2014. An Overview on Analysis Tools for Software Product Lines. In *Proc. Workshop on Software Product Line Analysis Tools (SPLat)*. ACM, 94–101.

[36] Marcílio Mendonça, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 761–762.

[37] Marcílio Mendonça, Andrzej Wąsowski, Krzysztof Czarnecki, and Donald Cowan. 2008. Efficient Compilation Techniques for Large Scale Feature Models. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 13–22.

[38] 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 *Proc. Int'l Conf. on Requirements Engineering (RE)*. IEEE, 243–253.

[39] Sarah Nadi, Thorsten Berger, Christian KÃdstner, and Krzysztof Czarnecki. 2015. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Trans. Software Engineering (TSE)* 41, 8 (2015), 820–841.

[40] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 907–918.

[41] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. 2018. Anomaly Analyses for Feature-Model Evolution. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 188–201.

[42] Michael Nieke, Christoph Seidl, and Thomas Thüm. 2018. Back to the Future: Avoiding Paradoxes in Feature-Model Evolution. In *Proc. Int'l Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution)*. ACM, 48–51.

[43] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM. To appear.

[44] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[45] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. 2011. Multi-Dimensional Variability Modeling. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 11–22.

[46] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany Ammar. 2013. Scalable Product Line Configuration: A Straw to Break the Camel's Back. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 465–474.

[47] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic Semantics of Feature Diagrams. *Computer Networks* 51, 2 (2007), 456–479.

[48] 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.

[49] Sergio Segura, José A. Galindo, David Benavides, José A. Parejo, and Antonio Ruiz-Cortés. 2012. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 63–71.

[50] Christoph Seidl, Ina Schaefer, and Uwe A. 2014. Capturing Variability in Space and Time with Hyper Feature Models. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 6, 6:1–6:8 pages.

[51] 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 (SQJ)* 20, 3-4 (2012), 487–517.

[52] Giovani Da Silveira, Denis Borenstein, and FlÃąvio S. Fogliatto. 2001. Mass Customization: Literature Review and Research Directions. *Int'l J. Production Economics* 72, 1 (2001), 1–13.

[53] 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.

[54] Sahil Thaker, Don Batory, David Kitchin, and William Cook. 2007. Safe Composition of Product Lines. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 95–104.

[55] 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.

[56] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract Features in Feature Modeling. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, 191–200.

[57] Thomas Thüm, Tim Winkelmann, Reimar Schröter, Martin Hentschel, and Stefan Krüger. 2016. Variability Hiding in Contracts for Dependent Software Product Lines. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 97–104.

[58] 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.

[59] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 178–188.

The High-Level Variability Language: An Ontological Approach

Angela Villota
CRI, Université Panthéon-Sorbonne
Paris, France
i2t, Universidad Icesi
Cali, Colombia
apvillota@icesi.edu.co

Raúl Mazo
CRI, Université Panthéon-Sorbonne
Lab-STICC, ENSTA Bretagne
France
GIDITIC, Universidad EAFIT
Medellín, Colombia
raul.mazo@univ-paris1.fr

Camille Salinesi
CRI, Université Panthéon-Sorbonne
Paris, France
camille.salinesi@univ-paris1.fr

ABSTRACT

Given its relevance, there is an extensive body of research for modeling variability in diverse domains. Regretfully, the community still faces issues and challenges to port or share variability models among tools and methodological approaches. There are researchers, for instance, implementing the same algorithms and analyses again because they use a specific modeling language and cannot use some existing tool. This paper introduces the High-Level Variability Language (HLVL), an expressive and extensible textual language that can be used as a modeling and an intermediate language for variability. HLVL was designed following an ontological approach, i.e., by defining their elements considering the meaning of the concepts existing on different variability languages. Our proposal not only provides a unified language based on a comprehensive analysis of the existing ones but also sets foundations to build tools that support different notations and their combination.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Software product lines**.

KEYWORDS

domain specific language, variability language, variability specification

ACM Reference Format:
Angela Villota, Raúl Mazo, and Camille Salinesi. 2019. The High-Level Variability Language: An Ontological Approach. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3307630.3342401

1 INTRODUCTION

Variability modeling is an extensively studied subject. Research in this subject includes several variability modeling languages that have been proposed in academia and industry [3]. Most research in the area focuses on feature-based modeling languages since the introduction of FODA [13]. However, other modeling approaches exist, variation point-based models [20], decision-based models [6], goal-oriented models [19], constraint-based languages [24] and industrial languages (e.g., Kconfig[28], Gears[15]) can be used to describe variability. These proposals have contributed to a universe of languages, notations, transformations, and tools supporting the creation of variability models.

Variability modeling languages are neither completely different nor completely the same. Indeed, these languages share most variability concepts such as variability units, and constraints, but also differ in concepts that are relevant to particular domains or modeling styles. For example, Figure 1 presents three variability models written in different languages: *FODA* [13], *Dopler* [6], *OVM* [20]. These models have (1) different structures (e.g., hierarchical, non-hierarchical); (2) different types of variability units (e.g., Boolean, non-Boolean); (3) heterogeneous rules (e.g., cross-tree constraints, visibility, and validity conditions); among others.

Currently, variability modeling relies upon existing domain-specific languages and modeling tools. These tools are developed and taught in-house and frequently are used only by the few people associated with the development team. This diversity of languages and tools causes lack of portability in models and interoperability issues between SPL engineering tools. One of the poor consequences is that modeling tools require numerous parsers and transformations that might cause expressiveness loss. To solve this gap, the Common Variability Language (CVL) was proposed as a standard language for variability modeling [12]. However, this initiative did not succeed, and the community still faces issues caused by the diversity of languages, dialects, and tools.

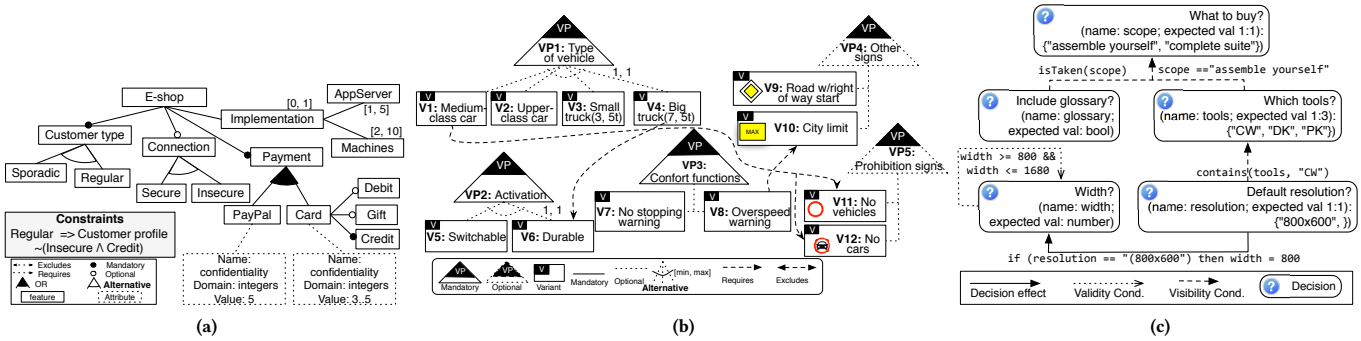
This paper presents our proposal for moving forward the initiative of defining a standard variability language. Particularly, this paper introduces: (i) a glossary of basic concepts from variability modeling languages, and (ii) a variability language able to describe these concepts and to work with/for/in combination with most languages and tools, the *High-Level Variability Language (HLVL).*

Developing a standard language can be an effort-intensive endeavor. Therefore, we followed an ontological approach for conceptualizing and structuring knowledge about variability modeling concepts. In this case, an ontological approach is favorable to determine a set of constructs to describe variability comprehensively and to define the characteristics of a language capable of representing constructs from different variability languages.

HLVL belongs to an ongoing project in which we envision a language capable of supporting concepts of many variability languages to reduce interoperability and sharing issues. The general idea is that variability models can be compiled into an intermediate

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342401



(a) (b) (c)

Figure 1: Variability models using three graphical languages. (a) FODA [13] with the extension proposed in [2]. (b) Orthogonal Variability Model (OVM) [20]. (c) Dopler Modeling Language (DoplerML) [6]

form and can be interpreted on other tools. In fact, if variability models are written in an intermediate language, such as HLVL, they could be integrated analyzed and configured into a single model in an integrated way. This research gives continuity to previous works [8, 17, 19, 24] that exploit the idea of relying on a variability language that unifies existing notations to provide genericity to the methods, techniques, and tools used for modeling, analysis, and configuration.

Section 2 the ontological approach used to design HLVL. Section 3 presents our proposal for a glossary of variability concepts. Section 4 introduces the syntax of HLVL and shows how HLVL supports different styles of variability modeling using examples. Sections 5 and 6 present the discussion and related work, respectively, and Section 7 concludes the paper concludes the paper with our final remarks.

2 DESIGNING HLVL FOLLOWING AN ONTOLOGICAL APPROACH

The ontological approach followed to structure the domain knowledge for designing the HLVL consisted of three steps depicted in Figure 2 and described below. Note that though this proposal unifies concepts from different variability languages, it does not subsume all variability languages, nor is our language capable of representing every single variability language. A broader ontological comparison would be necessary to produce such a language; which is a cumbersome and not scalable task.

Ontological analysis: in this step, we conducted an ontological analysis of the expressiveness of an initial version of the variability language. In this study, we used the variability patterns introduced by Asadi et al. [1] to determine the criteria for completeness and clarity from the ontological expressiveness perspective. The results showed that (1) the language closely represents the concepts in the ontological framework. However, some variability concepts should be integrated for obtaining a 100% level of completeness. (2) The language's high-level of abstraction impacts its clarity because several elements in the ontology are represented by the same language construct. A broader description of this analysis and its results are available in [27].

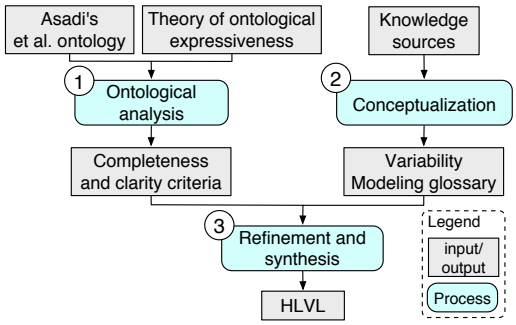


Figure 2: Process for designing HLVL

Conceptualization: this step consisted of the identification and review of research about variability modeling in literature. The studies selected for this step were gathered while conducting a systematic mapping study, an upcoming publication from the authors[1]. More particularly, the conceptualization included languages that (1) have been transformed to logic or constraint programming to automate analysis tasks; and (2) their semantics is formally defined, and were included in other conceptualization studies. Hence, this step included feature-oriented languages [5, 13, 14, 26]; variation point oriented languages [8, 12, 20, 22, 23]; and decision-oriented languages [4, 6, 7, 25]. Finally, the review included proposals introducing constructs for modeling complex variability relations, such as conditional and quantified constraints [8, 14, 21]. Based on the reviewed literature, a collection of variability modeling concepts were organized and structured in a glossary. Section 3 presents the result obtained in this step.

Refinement and synthesis: using the results from the previous steps, we proposed the characteristics of HLVL language. Section 4 presents the HLVL in detail.

3 VARIABILITY MODELING GLOSSARY

Variability languages enable the modeler to answer two questions about the product line to be modeled: *what does vary?* and *how does it vary?* [20]. These languages provide a collection of *constructs*

[1]Protocol available at http://bit.ly/2IVBde5

enabling the modeler (1) to identify and document the variable items in a product line; (2) to identify the set of possible options or variants associated to variable items in the system; (3) to identify the rules for determining how items can be combined into new configurations; and finally (4) to produce variability models. Language constructs in variability languages define the *variability units* and the *variability relations*.

3.1 Variability Units

Variability Units (VUs) are the key concept used to model variability in a language [4]. VUs represent variable items in a system or domain, that is, those aspects that must be chosen by the customer or engineer in a configuration process. For example, *features* are the VUs in feature models, *decisions* in decision models, and *variation points*, *variants* are the units in OVM models. VUs are characterized by their type and multiplicity.

Type. The type of a VU is defined by the number of variants it represents. VUs are Boolean when they are associated with exactly two options: *{selected, unselected}*, as in feature models. Non-Boolean VUs have more than two options, as *attributes* in attributed-based feature models, or *decisions* in decision models. Some languages such as Gears allow the declaration of complex data-types such as enumerations, sets, and records [15].

Multiplicity. It represents the number of instances of a VU that may appear in a configuration. Then, VUs are annotated using cardinalities in a UML style. For example, features with the interval $[m, n]$ have at least m and at most n instances in a configuration [5].

3.2 Variability relations

Variability relations determine the rules to select and recombine items into new products. Variability relations are often presented as dependencies or constraints and are usually denoted graphically (i.e., using arrows) or textually (i.e., logic formulas, OCL). Variability relations can be classified as follows:

Inclusion/exclusion rules. Are the set of basic rules for describing variability in a product line. In this set, we placed the rules for defining conditional inclusion/exclusion and commonalities.

Conditional inclusion/exclusion are rules for restricting the inclusion/exclusion of variable items given a condition. This condition can be simple as in FODA *requires* and *excludes* constructs where the inclusion/exclusion of a feature B is conditioned to the inclusion of a feature A. Languages such as CO-OVM[8], and extended-feature models in [14] allow the usage of complex expressions to condition the inclusion/exclusion of variable items using logical expressions.

Commonality. Rules for defining items that always appear in any configuration. This rule is implicit in some languages (e.g., root feature in feature models, mandatory variation points in OVM) or inexistent (e.g., decision models). This is different from calculating the set of core items, an analysis operation that requires extra processing.

Hierarchy - Decomposition. In decomposition relations, one item plays the role of parent, and the other is the child. This parent-child relation imposes a constraint in the configuration because no child can be part in a configuration without the inclusion of its parent. There are two types of decompositions:

On-to-one decompositions relate pairs of items. There are two types of decompositions: *mandatory* and *optional*. In mandatory decompositions, the child is included in all products in which its parent appears. Instead, in optional decompositions, the child can be optionally included in all products in which its parent appears.

One-to-many decompositions relate one parent and a group of children. This relation restricts the minimum and the maximum number of children that may be included in a configuration when their parent is selected.

Hierarchy - visibility. Visibility rules condition the availability of other variability items. Visibility relations are considered a type of hierarchical relations because they are used to compartmentalize items, as in different views, e.g., for different stakeholders [4]. Visibility relations are a common construct in decision-based languages such as Dopler [6].

Constraint expressions. Constraint expressions are used to include complex rules between variable items in a product line model. This rules can be composed using relational, arithmetic and global operators among others. These constraint expressions are often used to specify extra-functional information or to include contextual rules. For instance, the extended-feature models in [14] allow the inclusion of expressions and operators for adding constraints between feature's attributes and instances.

4 THE HIGH-LEVEL VARIABILITY LANGUAGE

HLVL is a variability modeling language that addresses the following three requirements:

Rq1. HLVL provides constructs to model the concepts in the variability modeling glossary to describe variability comprehensively.

Rq2. HLVL specifies variability models in different approaches such as feature-oriented, variation point oriented, and decision-oriented modeling.

Rq3. HLVL's syntax should be understandable for humans to create and edit models, but also should be formally defined to be generated and interpreted by modeling tools.

The following subsections present an overview of the HLVL syntax (see Section 4.1) and further examples of the HLVL's usage (see Section 4.2).

4.1 Syntax

Variability models in HLVL are defined in terms of constructs called **elements**, **variants**, and **variability relations**. These constructs map the concepts defined in the variability modeling glossary previously presented in Section 3. The collection of elements, variants, and variability relations describing a model in HLVL conform a **script**. Scripts in HLVL start with the keyword `model` followed by an identifier. Each block of the script also starts with a keyword (i.e., `elements`, and `relations`). A simple E-shop product line adapted from [22] (see Figure 1a) serves us to illustrate the HLVL's syntax (see Table 1). This example describes a basic scenario that will grow as we introduce language constructs. These constructs are formally defined using the BNF grammar summarized in Table 2.

Table 1: Running Example

The online-shopping product line is a collection of similar e-commerce web applications. All products in the E-shop domain must have a module to handle the customer type, a module for manage the payment, and optionally, some modules for managing users' connection. In this product line, the payment method may be provided by PayPal services, or card. Additionally, the system must guarantee a secure connection when the transaction is performed by a regular customer o when the payment is performed using a credit card. Finally, each payment module has a confidentiality level that represents the privacy level of payment details. The confidentiality levels range from one to five.

4.1.1 Elements and Variants. Elements are the variability unit in HLVL. Elements in HLVL are typed, and optionally include a keyword to define attributes or comments introduced by the modeler. HLVL supports `boolean`, `integer`, and `symbolic` data types. Each element is associated with a set of variants that represents the available choices. The configuration process selects exactly one choice from the set of variants (i.e., enumeration). The variants associated to an element are declared using intervals or lists of values regarding the data type. The following example shows the declaration of a group of Boolean elements and a symbolic element. As shown in the example, in HLVL, Boolean variants do not require an explicit declaration (syntactic sugar).

```
model eShop
elements:
  boolean connectionType, secureConnection, insecureConnection,
      payment, payPal
  symbolic customerType variants: ['sporadic', 'regular']
      comment: {"This element represents the customer type"}
```

Note that HLVL's identifiers can be composed in a flexible programming language fashion. Then, the only rules regarding identifiers are that they cannot start with digits, special characters, or contain spaces.

Attributes. Elements can be used to represent attributes. In HLVL, we differentiate an attribute from a regular element using the keyword `att` in the element declaration. For example, to define an integer attribute representing confidentiality in HLVL, we write the following:

```
att integer confidentiality variants: 1..5
att integer confBounded is 2
```

The example also shows the definition of `confBounded` as a bounded attribute to a value by the keyword `is`. We included this definition to support the simplification of attributes introduced by some tools.

Multiplicity. Elements in HLVL can have multiple instances with local semantics as described in [18]. Syntactically, multiplicities are declared as properties for dependency relations as we explain below.

4.1.2 Variability Relations. Variability relations in HLVL can be used for (1) defining inclusion/exclusion rules; (2) describing hierarchies; (3) constraining the visibility of other variability relations; and (4) including complex expressions such as arithmetic, logic, and relational. Table 2 presents the syntactic rules for HLVL's variability relations (i.e., $R7 - R17$). As shown in this table, all variability relations in HLVL contain an identifier for referencing.

Table 2: Syntax for variability relations in HLVL

⟨*element*⟩ ::=	{`att`}? ⟨*data_type*⟩ E_1 `variants:` ⟨*variants*⟩ {`comment:String`}? |	($R1$)
	{`att`}⟨*data_type*⟩ E_1 `is` ⟨*value*⟩	($R2$)
⟨*variants*⟩ ::=	⟨*integer*⟩..⟨*integer*⟩ |	($R3$)
	[⟨*value*⟩{,⟨*value*⟩}*]	($R4$)
⟨*data_type*⟩ ::=	`boolean` | `integer` | `symbolic`	($R5$)
⟨*value*⟩ ::=	`true` | `false` | {0..9}+ | '⟨*string*⟩'	($R6$)
⟨*sentence*⟩ ::=	⟨*identifier*⟩ : ⟨*relation*⟩	($R7$)
⟨*relation*⟩ ::=	`common`($E_1, E_2, \ldots, E_k$) |	($R8$)
	`mutex`(E_1, E_2) |	($R9$)
	`mutex`(⟨*expression*⟩,$E_1, \ldots, E_k$) |	($R10$)
	`implies`(E_1, E_2) |	($R11$)
	`implies`(⟨*expression*⟩,$E_1, \ldots, E_k$) |	($R12$)
	`decomposition`(P,[$C_1, C_2, \ldots, C_k$], $[m,n]$) |	($R14$)
	`group`(P,[$C_1, C_2, \ldots, C_k$], $[m,n]$) |	($R15$)
	`visibility`(⟨*expression*⟩,[$R_1, \ldots, R_k$])	($R16$)
	`expression`(⟨*expression*⟩)	($R17$)

Commonality. HLVL provides a construct to declare common elements in a product line explicitly. In the running example, the modules for handling the customer type and the payment are always part of an E-shop. In HLVL, this is expressed as follows:

```
com1: common(customerType, payment)
```

Inclusion/Exclusion relations. HLVL provides different constructs to describe inclusion and exclusion rules in particular, *constraint expressions*. Constraint expressions in HLVL are useful for including complex rules between elements in the variability model using logic, relational, arithmetic, and global operators. These complex rules are written in the HLVL's expressions language (cf. Table 3), that is also used to write the conditions in other variability relations. For example, to restrict the confidentiality levels of the payment by card module to be between 3 and 5, we write the following relation in HLVL:

```
exp1: expression(3 <= card.confidentiality AND
                 card.confidentiality <= 5)
```

Conditional exclusion/exclusion relations can optionally be described using language constructs. To this purpose, HLVL provides the keywords `mutex` and `implies`. Through language constructs, HLVL supports two types of conditional exclusion: *mutual exclusion* and *guarded exclusion*. Consider the following example:

```
m1: mutex(creditCard, insecureConnection)
m2: mutex(customerType='sporadic',[ giftCard , creditCard ])
```

Here, $m1$ represents the mutual exclusion of the credit card payment and insecure connection. Then, these two elements cannot be part of the same configuration. Also, the guarded exclusion $m2$ defines a condition to exclude the payment by gift card and debit card for sporadic customers. Guarded exclusion may have complex conditions using HLVL's expressions language. Then, whenever the condition is satisfied the group of elements won't be included in a configuration.

Similarly, HLVL supports *implication* and *guarded implication* using constructs as follows.

Table 3: Syntax of the expressions language

⟨*expression*⟩ ::=	~ ⟨*boolExp*⟩ | ⟨*boolExp*⟩ | ⟨*assignExp*⟩
⟨*boolExp*⟩ ::=	⟨*boolVal*⟩ | ⟨*boolExp*⟩ ⟨*logicOp*⟩ ⟨*boolExp*⟩ | ⟨*relational*⟩
⟨*relational*⟩ ::=	⟨*arithmetic*⟩ ⟨*relationalOp*⟩ ⟨*arithmetic*⟩
⟨*arithmetic*⟩ ::=	⟨*numericVal*⟩ | ⟨*numericVal*⟩ ⟨*arithmeticOp*⟩ ⟨*numericVal*⟩
⟨*boolVal*⟩ ::=	⟨*name*⟩ | `true` | `false`
⟨*numericVal*⟩ ::=	⟨*name*⟩ | ⟨*integer*⟩
⟨*logicOp*⟩ ::=	`AND` | `OR` | `=>` | `<=>`
⟨*RelationalOp*⟩ ::=	`=` | `!=` | `>` | `>=` | `<` | `<=`
⟨*arithmeticOp*⟩ ::=	`+` | `-` | `*` | `/`
⟨*name*⟩ ::=	⟨*name*⟩.⟨*name*⟩ | ⟨*identifier*⟩

```
imp1:implies(payPal, secureConnection)
imp2: implies(customerType='regular', [secure, customerProfile])
```

In this example, we use *imp*1 to represent that the inclusion of PayPal payment implies the use of the secure connection (i.e., requires). Also, *imp*2 represents the inclusion of the modules for handling secure connection and customer's profile, conditioned to the selection of a regular customer. Conditions in guarded inclusions are written using constraint expressions.

Hierarchy-Decomposition. Although HLVL is not a language where hierarchical relations are essential for composing models, it offers a set of constructs to describe one-to-one (parent-child), and one-to-many (parent-children) decompositions.

One-to-one decompositions in HLVL contain the keyword `decomposition` followed by the names of the parent and the child element together with a cardinality $[m, n]$. This cardinality is a *multiplicity annotation* and is used to bound the number of instances of the child element. Decompositions of type mandatory and optional can be considered special cases with the cardinalities $[1, 1]$ and $[0, 1]$, respectively. In the running example, the relations stating that the gift-card and debit-card modules are optional and the credit-card module is mandatory are written in HLVL as follows:

```
dc1: decomposition(card, [giftCard, debitCard],[0,1])
dc2: decomposition(card, [creditCard],[1,1])
```

To illustrate dependencies with cardinality $[m, n]$, let us now make an addition to the running example: Suppose that the E-shop product line is implemented using between one and five application servers (e.g., Glassfish, Tomcat, Jetty, etc) supported by minimum two and maximum ten machines (see Figure 1a). In HLVL, this is written as follows:

```
dc3: decomposition(implementation, [appServer], [1,5])
dc4: decomposition(implementation, [machines], [2,10])
```

Decompositions with cardinality $[0, 1]$ are used to associate Boolean elements to one or more attributes. Let's extend the example, including an attribute for the type of security certificate in the payment modules. The association of elements to attributes in HLVL is written as follows.

```
a1:decomposition(payPal,[ confidentiality , certificateType ],[1,1])
a2:decomposition(card,[ confidentiality , certificateType ],[1,1])
```

The inclusion of these relations enable the qualified names `payPal.confidentiality`, `payPal.certificateType`, `card.confidentiality`, and `card.certificateType` to differenciate each attribute. Also, note that elements cannot represent attributes and have multiplicities at the same time.

one-to-many decompositions contain the `group` construct, the identifier of the parent, the children identifiers enclosed in brackets followed by an interval representing the cardinality. This cardinality is used to specify the minimum and the maximum number of children that can appear in a product. For example, in the E-shop product line, the variability in the payment method can be modeled using a group relation as follows:

```
g1: group(payment, [payPal, card], [1,*])
```

In this example, the cardinality `[1,*]` denotes that at least one, and at most the number of children can be selected.

Visibility. Visibility relations in HLVL are rules to condition the availability (i.e., hide) of a group of elements and their relations with similar semantics than visibility rules in decision models [7]. These relations are declared starting with the keyword `visibility` followed by a constraint expression and the identifiers of the elements this condition hides. For example, let's imagine that the implementation characteristics of the E-shop are associated with the company business (i.e., service seller or product seller). Then, elements `implementation`, `appServer`, and `machines` will be visible only if the company commercializes services. We can represent this in HLVL as follows:

```
v1: visibility (productType = 'services', [implementation,
    appServer, machines])
```

4.2 Other Examples in HLVL

The following subsections show how specific constructs of other notations are represented using HLVL. First, the excerpt of the model of the Radio Frequency Warner system (RFW) product line taken from [23] and written in OVM (Figure 1b). Second, the excerpt of the dopler model describing the variability of the Dopler tool suit taken from [16] (Figure 1c). The examples at their full extent are available at https://github.com/angievig/Coffee/tree/master/HLVL/Examples/MODEVAR.

4.2.1 Modeling Variation Point Languages in HLVL. Variation Points (VP) and variants can be modeled using Boolean elements in HLVL. For example, the variation point VP5 in Figure 1b representing the prohibition signs and its variants, the no vehicles sign (V11), and no cars sign (V12) can be modeled in HLVL as follows:

```
boolean VP5 comment:{"Prohibition signs"}
boolean V11 comment:{"No vehicles"}
boolean V12 comment:{"No cars"}
```

In this example, we used the VP's identifier to name the element in HLVL and the keyword `comment` to include the extra information in the diagram.

Mandatory VPs, that is variation points that must always be bounded are declared using the `common` construct. In the RFW, `VP1`, `VP2`, and `VP3` are mandatory VPs, this is expressed in HLVL as follows:

```
c1: common(VP1, VP2, VP3)
```

The links between VPs and variants (i.e., mandatory, optional, and alternative) are one-to-one and one-to-many decompositions.

In HLVL, mandatory and optional links are represented using the **decomposition** construct. Alternative $[m, n]$ links are represented using the **group** construct. For instance, the optional relation between VP5 and V11, V12, and the alternative relation between VP1 and V1, V2, V3, V4 is written in HLVL as follows:

```
d1: decomposition(VP5, [V11, V12], [0,1])
d2: group(VP1, [V1, V2, V3, V4], [1,1])
```

Alternatively, VPs and variants linked by alternative relations with cardinality [1..1] can be represented in HLVL using a symbolic element for the VP and symbolic values for the variants in the group. For example, VP2 and V5, V6 can be written in HLVL as follows:

```
symbolic VP2 variants: ['V5', 'V6'] comment:{"Activation"}
```

Constraints in OVM models can me represented using the **implies** and **mutex** constructs. In the example, the implication (requires) between V8 and V10 is expressed as follows:

```
imp2: implies(V8, V10)
```

When the modeler choose to represent alternative [1..1] relations using symbolic elements, the constraints can be represented with constraint expressions and guarded implications in HLVL. Let's imagine that VP1 and VP2 are symbolic symbols, then the implications between pairs (V4, V6), (V1, V11), and (V1, V12) can be written in HLVL as follows:

```
exp1: expression(VP1 ='big truck' => VP2 ='durable')
imp1: implies(VP1 ='medium-class car', [V11,V12])
```

Attributes and complex constraints in the CO-OVM style [8] are represented in HLVL using constraint expressions and conditional implications.

4.2.2 Modeling Decision Models in HLVL. Decisions with cardinality 1 : 1 can be modeled using elements, their data types, and comments. To illustrate this, the scope and glossary decisions in the example are represented in HLVL as follows:

```
symbolic scope variants: ['assemble yourself', 'complete suite']
        comment: {"What to buy?"}
boolean glossary comment:{"Include glossary?"}
```

Additionally, decisions with cardinality $1 : N$ are represented using Boolean elements and a **group** relation. For example, the decision tools, its three variants, and its cardinality are written as follows in HLVL:

```
boolean tools, confWizard, decisionKing, projectKing
g1:group(tools ,[confWizard,decisionKing, projectKing ], [1,3])
```

Visibility conditions in decision models are modeled in HLVL using the **visibility** construct. In the example, the decision about the resolution is visible if the user decides to include the configuration wizard tool. Also, the glossary will be included after a decision about the scope is taken. These visibility conditions are expressed in HLVL as follows:

```
vis1: visibility (confWizard=true, [ resolution ])
vis2: visibility ( entailed (scope), [glossary ])
```

The function **entailed** is used to determine if the value of an element is already decided.

Decision effects in dopler models describe dependencies between decisions as rules triggering values for other decisions. For example, the rule determining that the selection of the resolution triggers the value of the width is written using constraint expressions in HLVL as follows:

```
e1: expression (( resolution ='800x600') => width = 800))
```

Validity conditions are the rules restricting the range of the values which can be assigned to a decision. In HLVL, these rules are written using constraint expressions. For example, the validity condition restricting the width as a number between [800, 1680] is written in HLVL as follows: the

```
val1: expression (width >= 800 AND width <= 1680)
```

5 DISCUSSION

5.1 Intermediate Language for Variability

One of the main characteristics of HLVL is that it contains constructs for comprehensively modeling variability concerning the concepts in the variability glossary presented in Section 3. Hence, HLVL can be used as used (1) as a specification language to create variability models; or (2) as an intermediate representation of models specified in other variability languages. Here, we borrow the concept of intermediate language from the compilers' domain. In this domain, intermediate languages are used to produce intermediate representations during the process of translating a source program into target code. Many compilers generate an explicit low-level or machine-like intermediate representation, which can be thought of as a program for an abstract machine, as in the case of the Java language.

The usage of an intermediate language for variability is a viable alternative to the interoperability problem because written in such a language, variability models can be easily shared or distributed. Then, modeling tools should be able to export and import models in the intermediate language, so modelers do not have to learn a new variability language, then, modeling tools can be used as they are today. An intermediate language for variability can be to variability modeling tools as the BibTeX format is for reference tools. That is, these managing references applications (e.g., Mendeley, Zotero, etc.) have their own formats and styles for managing references. Yet, these applications are also capable of importing and exporting BibTeX formats. Even electronic databases, for example, ACM data library, IEEE Xplore, Springer, Science Direct, citeSeer, etc. have their own way to store and display references. However, these electronic databases provide an option for downloading or exporting references in the BibTeX format. Moreover, if the rare cases that a publication has not an available a BibTeX format, it is possible to define its BibTeX because the language' syntax is simple. Also, there exits examples and documentation for the BibTeX notations are publicly available.

5.2 Many Languages, One Representation

To ensure the flexibility of the language, HLVL has syntactical elements for modeling Boolean and non-Boolean variability supporting the description of simple and complex variability models. Besides, HLVL supports different styles of variability modeling. As shown in the examples above, HLVL can be used to specify FODA models, attribute-based feature models, cardinality-based feature models, variation-point oriented models or even decision-based

models. In the conducted literature review, we observed that variability models are often enriched with ad-hoc constraints to gain expressiveness. These approaches contribute to the proliferation of new dialects and language extensions. Considering that the transformation of the base language into an HLVL model is viable, the new constraints can be introduced in HLVL. Then, HLVL can be a standard language to add variability relations not supported by current notations to enhance variability models without increasing the variability of variability languages.

The support of Boolean and non-Boolean variable items, the capability of model different variability languages, and the potential capability to enhance variability models let us envision HLVL as a viable language for integrating variability models described in different languages. Either written in the same editor in HLVL or created in different modeling tools, models from different sources can be integrated to be analyzed or configured.

5.3 What is Next for HLVL?

This paper presented the syntax, semantics, and usage scenarios of HLVL. However, the full design of the language comprises the formal definition of semantics and other syntactic aspects such as well-formedness rules and lexical syntax. To this purpose, we will consider the guidelines for defining modeling languages proposed by Harel and Rumpe in [11]. Also, we will examine the formal semantics for the variability languages supported by HLVL [7, 18, 26]. We will study carefully the semantics of visibility and multiplicity relations, as well as the effects in the configuration semantics produced by these relations.

Then, the next step in our research consists in the evaluation of the language focusing on expressiveness. This evaluation will measure the language's expressiveness from two different points of view. First, we will conduct a new ontological analysis to verify that this new proposal solves the issues reported in our previous work [27]. We are aware that the fact inclusion/exclusion relations can be represented by more than one language construct produces construct redundancy, which is one of the defects in conceptual modeling languages. However, we consider that this defect is preferable to the ambiguities created by having one construct mapping many variability concepts. These ambiguities may interfere in the transformation process to obtain the HLVL representation of a model initially written in another language.

In the second part of the evaluation, we will show that HLVL's constructs map the constructs of variability languages implemented by particular modeling tools. Section 4 presented three examples used to illustrate how concepts from different academic variability modeling languages can be represented using HLVL. In this evaluation, we are interested in providing transformation rules and algorithms to produce HLVL models using as input the formats produced by tools implementing those academic languages. The first step in this direction was the implementation of a concept-proof including an editor for HLVL models and the Java tools for translating variability models specified in two different tools (Code available at https://github.com/angievig/CoffeeProofOfConcept). This implementation was used to demonstrate the feasibility of the usage of HLVL as a modeling and intermediate language. We plan to apply this proposal in the reengineering of the VariaMos SPL tool suite.

6 RELATED WORK

In previous works [8, 17, 19, 24] we have developed the idea to provide full *genericity* to methods, techniques, and tools for variability modeling, analysis, and configuration. First, Salinesi et al. [24] proposed to use a constraint programming language to represent variability. This proposal relied on the expression power of constraint programming. However, the benefits of this language do not compensate the drawbacks in the design given the language's lack of usability and readability. At some point, to use this language resembled replacing a programming language by assembly language: regardless its benefits, to work with large scale assembly programs without a higher level, more abstract language is an unfeasible task.

More recent works [17, 19] introduced different levels of metamodels to provide a high-level view of the constraint language that serves as generic language. This proposal is fully implemented in the current VariaMos tool suite supporting different variability languages and providing tools for extensions. However, this is a complex approach with a lack of usability and poor tool performance. Dumitrescu et al. [8] developed a variant of SysML to address the design of cyber-physical systems considering industry standards. The variability language proposed in this work contains various constructs that are not relevant to variability specification. This paper, presents an ontological approach introducing HLVL, an agnostic variability representation that serves as modeling and intermediate language. We followed a compiler's approach where the language provides a formal syntax. This approach eases the generation of HLVL code from other tools.

Eichelberger and Schmid report a certain trend in product line engineering towards textual variability modeling languages [9]. The survey and analysis of textual variability languages presented in their work characterizes eleven textual languages, including their own proposal the INDENICA Variability Modeling Language (IVML). Among the results, the authors report that most textual variability languages support feature-oriented modeling and less frequently, variation point oriented modeling. In contrast, our proposal supports feature-oriented modeling, variation-point oriented modeling, and decision-based modeling. In consequence, the HLVL allows the creation of models in any of these styles of modeling. Moreover, to combine in a model constructs from different approaches that were traditionally exclusive to a set of modeling languages.

Galindo et al. [10] present an approach to ease the integration of variability models specified using different modeling styles, variability languages, and tools to perform configuration. They introduce the Invar approach to provide the user with a configuration tool that hides the different models, their semantics, and internal representation. Then, the configuration is performed by different tools and is orchestrated by an API that manages the communication between the configurator and the analysis and configuration tools. Alternatively, our proposal considers the integration of variability models using HLVL as an intermediate language to perform analysis and configuration operations.

7 CONCLUDING REMARKS

Migrating or integrating models built with different languages is challenging because many concepts and forms are not consistent among them. To define a unified language, we have been applying an ontological approach, i.e., we have analyzed feature-oriented, variation point oriented, and decision-oriented languages to define a glossary of concepts and propose a unified language based on this glossary. This paper introduces the High-Level Variability language (HLVL), a unified variability language defined following our ontological approach. Here, we presented the HLVL using an example containing complex rules considering Boolean and non-Boolean elements, attributes, multiplicities, and constraint expressions. Also, we show how HLVL supports different styles of variability modeling using two examples in different languages.

HLVL is a declarative language with a formally defined syntax that resembles programming languages. The formal definition of HLVL's syntax eases the code generation from other variability languages. Also, being a programming-like language, HLVL is a more human-readable language than other markup languages. However, we consider that the concrete syntax presented in this paper may evolve as a consequence of further validation and evaluation. We believe that the ontological approach in this research and the resulting unified language are viable alternatives to the interoperability issues evidenced by the product line community. This research contributes to reducing the lack of consensus in the concepts that should be included in variability languages. Also, it contributes to the proposal of a standard format useful for the portability of variability models.

Further discussion is required in this subject since the discussion was oriented from the modeling perspective, more particularly from an expressiveness perspective. However, the discussion should also focus on the characteristics of the variability language that ease the analysis and extraction of information from variability models. This other perspective and further evaluation of the HLVL are part of our ongoing project about the application of constraints for variability modeling and reasoning.

REFERENCES

[1] Mohsen Asadi, Dragan Gasevic, Yair Wand, and Marek Hatala. 2012. Deriving Variability Patterns in Software Product Lines by Ontological Considerations. In *Conceptual Modeling – ER*, Vol. 7532 LNCS. Springer Berlin Heidelberg, Berlin, Heidelberg, 397–408. https://doi.org/10.1007/978-3-642-34002-4_31

[2] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. In *Advanced Information Systems Engineering*. Springer, 491–503. https://doi.org/10.1007/11431855_34

[3] 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 (2013), 1611–1640. https://doi.org/10.1109/TSE.2013.34

[4] 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 Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. ACM, NY, USA, 173–182. https://doi.org/10.1145/2110147.2110167

[5] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice* 10, 1 (jan 2005), 7–29. https://doi.org/10.1002/spip.213

[6] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. 2011. The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. *Automated Software Engineering* 18, 1 (mar 2011), 77–114. https://doi.org/10.1007/s10515-010-0076-6

[7] Deepak Dhungana, Patrick Heymans, and Rick Rabiser. 2010. A Formal Semantics for Decision-oriented Variability Modeling with DOPLER. In *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings*. 29–35.

[8] Cosmin Dumitrescu, Patrick Tessier, Camille Salinesi, Sebastien Gérard, Alain Dauron, and Raul Mazo. 2014. Capturing Variability in Model Based Systems Engineering. In *Complex Systems Design & Management*. Springer International Publishing, 125–139. https://doi.org/10.1007/978-3-319-02812-5_10

[9] Holger Eichelberger and Klaus Schmid. 2015. Mapping the Design-space of Textual Variability Modeling Languages: A Refined Analysis. *Int. J. Softw. Tools Technol. Transf.* 17, 5 (Oct. 2015), 559–584. https://doi.org/10.1007/s10009-014-0362-x

[10] José A. Galindo, Deepak Dhungana, Rick Rabiser, David Benavides, Goetz Botterweck, and Paul Grünbacher. 2015. Supporting distributed product configuration by integrating heterogeneous variability modeling approaches. *Information and Software Technology* 62 (2015), 78 – 100. https://doi.org/10.1016/j.infsof.2015.02.002

[11] D. Harel and B. Rumpe. 2004. Meaningful modeling: what's the semantics of "semantics"? *Computer* 37, 10 (oct 2004), 64–72. https://doi.org/10.1109/MC.2004.172

[12] Øystein Haugen. [n. d.]. Common variability language (CVL) - OMG revised submission. OMGdocumentad/2012-08-05,2012

[13] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Software Engineering Institute, Carnegie Mellon University.

[14] Ahmet Serkan Karataş, Halit Oğuztüzün, and Ali Doğru. 2013. From extended feature models to constraint logic programming. *Science of Computer Programming* 78, 12 (dec 2013), 2295–2312. https://doi.org/10.1016/j.scico.2012.06.004

[15] Charles W. Krueger. 2007. BigLever Software Gears and the 3-tiered SPL Methodology. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA '07)*. ACM, New York, USA, 844–845. https://doi.org/10.1145/1297846.1297918

[16] Raúl Mazo, Paul Grünbacher, Wolfgang Heider, Rick Rabiser, Camille Salinesi, and Daniel Diaz. 2011. Using constraint programming to verify DOPLER variability models. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems - VaMoS '11*. ACM Press, New York, USA, 97–103. https://doi.org/10.1145/1944892.1944904

[17] Raúl Mazo, Camille Salinesi, Daniel Diaz, Olfa Djebbi, and Alberto Lora-Michiels. 2012. Constraints: The Heart of Domain and Application Engineering in the Product Lines Engineering Strategy. *International Journal of Information System Modeling and Design* 3, 2 (2012), 33–68. https://doi.org/10.4018/jismd.2012040102

[18] Raphael Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. 2011. A Formal Semantics for Feature Cardinalities in Feature Diagrams. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '11)*. ACM, New York, USA, 82–89. https://doi.org/10.1145/1944892.1944902

[19] Juan C. Muñoz-Fernández, Gabriel Tamura, Irina Raicu, Raúl Mazo, and Camille Salinesi. 2015. REFAS: a PLE approach for simulation of self-adaptive systems requirements. In *Proceedings of the 19th International Conference on Software Product Line - SPLC '15*. ACM Press, New York, USA, 121–125. https://doi.org/10.1145/2791060.2791102

[20] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/3-540-28901-1

[21] Clément Quinton, Daniel Romero, and Laurence Duchien. 2013. Cardinality-based feature models with constraints. In *Proceedings of the 17th International Software Product Line Conference on - SPLC '13*. ACM Press, New York, New York, USA, 162. https://doi.org/10.1145/2491627.2491638

[22] Fabricia Roos-Frantz, David Benavides, and Antonio Ruiz Cortés. 2010. Automated Analysis of Orthogonal Variability Models using Constraint Programming.. In *JISBD*. 269–280.

[23] Fabricia Roos-Frantz, David Benavides, Antonio Ruiz-Cortés, André Heuer, and Kim Lauenroth. 2012. Quality-aware analysis in product line engineering with the orthogonal variability model. *Software Quality Journal* 20, 3-4 (2012), 519–565.

[24] Camille Salinesi, Raul Mazo, Daniel Diaz, and Olfa Djebbi. 2010. Using Integer Constraint Solving in Reuse Based Requirements Engineering. In *Proceedings of the 2010 18th IEEE International Requirements Engineering Conference (RE '10)*. IEEE Computer Society, Washington, DC, USA, 243–251. https://doi.org/10.1109/RE.2010.36

[25] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. 2011. A comparison of decision modeling approaches in product lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems - VaMoS '11*. ACM Press, New York, USA, 119–126. https://doi.org/10.1145/1944892.1944907

[26] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic Semantics of Feature Diagrams. *Comput. Netw.* 51, 2 (2007), 456–479. https://doi.org/10.1016/j.comnet.2006.08.008

[27] Angela Villota, Raúl Mazo, and Camille Salinesi. 2018. On the Ontological Expressiveness of the High-Level Constraint Language for Product Line Specification. In *System Analysis and Modeling. Languages, Methods, and Tools for Systems Engineering*. Springer, Copenhagen, 46–66. https://doi.org/10.1007/978-3-030-01042-3_4

[28] Zippel and Contributors. [n. d.]. kconfig-language.txt. https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt

Towards a New Repository for Feature Model Exchange

José A. Galindo and David Benavides
Universidad de Sevilla
Sevilla, Spain
{jagalindo,benavides}@us.es

ABSTRACT

Feature models are one of the most important contributions to the field of software product lines, feature oriented software development or variability intensive systems. Since their invention in 1990, many feature model dialects appeared from less formal to more formal, from visual to textual, integrated in tool chains or just as a support for a concrete research contribution. Ten year ago, S.P.L.O.T. a feature model online tool was presented. One of its most used features has been the ability to centralise a feature model repository with its own feature model dialect. As a result of MODEVAR, we hope to have a new simple textual feature model language that can be shared by the community. Having a new repository for that language can help to share knowledge. In this paper we present some ideas about the characteristics that the future feature model repository should have in the future. The idea is to discuss those characteristics with the community.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; *Requirements analysis*; *Software design engineering*; *Software implementation planning*;

KEYWORDS

feature model repository, characteristics, variability, requirements

ACM Reference Format:
José A. Galindo and David Benavides. 2019. Towards a New Repository for Feature Model Exchange. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3307630.3342405

1 INTRODUCTION

Feature models are one of the most important contributions since software product lines flourished [6] back in 1990. There are commercial and academic software product line development tools that use feature models as a cornerstone language. Most of these tools use their own feature modelling dialect. Often, the semantics are quite similar while there are differences among the concrete syntax used to describe feature models. There are many textual and graphical feature models dialects that can be found in the literature [5]. One of the ideas of the MODEVAR workshop is to find a common, simple feature model language that can be used to share knowledge among researchers.

Ten years ago S.P.L.O.T., a feature model online tool, was presented [8]. One of the most used characteristics of S.P.L.O.T was the feature model repository that allows to share models among researchers and practitioners. However, as other tools, S.P.L.O.T also defined its own feature modelling language and the interoperability with other tools was not always straightforward.

There is a strong effort around the world for what is called *Open Science* [9] that promotes – among other principles– the reproducibility of experiments and information sharing. We believe that the software product line community has to put some effort to follow this line. Although S.P.L.O.T. has played its role during these years, we think that a new feature model repository has to be set up with new characteristics, technologies and open science spirit.

In this paper we present some ideas about the elements that a feature model repository could have in the future. The idea is to discuss those elements among the community to join forces to set up a shared feature model repository for the next decades that can complement the new feature model language.

2 REPOSITORY CHARACTERISTICS

Following, we enumerate and explain some of the characteristics that we envision in a new feature model repository using the technique of *user stories* [2]. The characteristics were defined observing S.P.L.O.T. and other repositories such as Zenodo [1] and after some brainstorming discussions among the authors. We present the user stories following the template of *"As a [persona], I [want to], [so that]."* Then, we justify the potential importance of the user stories.

(1) *As a researcher, I want to upload models to the repository, so that my models are available for others.* This is one of the most basic features. It seems to be simple but some complexity can appear. For example, a syntax checker could be added to ensure that the uploaded models are syntactically correct. This raised an additional user story as described below. Also, a cross check validation could be implemented to approve or deny uploads according to some criteria. Also, depending on other user stories, some additional information to upload the models can vary.
(2) *As a researcher, I want to syntax check my models before uploading in the repository, so that my models are syntax error free.* This will clearly depend on the abstract and concrete syntax defined for the language.
(3) *As a researcher or user, I want to download models from the repository, so that I can replicate experiments or use existing models.* This is also a basic feature but could also face some

[1] https://zenodo.org/

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342405

complexity during its development. For instance, the way models are downloaded could vary from one by one to batch downloads. Designing a good solution for downloading models seems to be important to have a useful repository.

(4) *As a researcher, I want to generate a universal identifier for my models in the repository, so that my models are citable and easily identified.* A possible interesting characteristic of the repository could be to have a citable identified in the form of a DOI or something similar. The DOI could correspond with the DOI of a related research paper or could be a dedicated DOI just for the model or even a set of models. This is an interesting feature but may have a lower priority than the former ones.

(5) *As a researcher or user, I want to manage versions of my models, so that I can compare different versions.* This was a hardly discussed story among the authors. We see a potential of having models or benchmark versions but at the same time we see that this can be a characteristic that could make the repository more complex to maintain.

(6) *As a user, I want to search for models in the repository, so that I can find the models I am interested in.* A good search engine can help find models among the ones in the repository. Other search types could be implemented such as tag–based or author–based.

(7) *As a user, I want to display models in the repository, so that I can have a glance at the model in the case I want to use it.* Feature models of the repository could be used for benchmarking purposes but also for teaching or dissemination. In some cases, it could be useful to display the model. It has to be decided how to visualise the model. A textual visualisation will be available but other visual syntax could be defined/discussed.

(8) *As a user or researcher, I want to know some indicators about the models in the repository such as ratings or number of downloads, so that I can compare models.* This can allow to see how many times a model has been downloaded or rated. This could also help on the search user story described above.

(9) *As a developer or researcher, I want to have an API that can interact with the repository, so that I can programmatically access repository's information.* An interesting feature of a repository would be that the content could be accessed programatically in the form of an API. This could allow a developer to call a method to download some models with some characteristics for example such as a given number of features.

(10) *As a user, I want to have recommendations according to my profile, so that I can do better model selections.* If metrics of the models are stored according to downloads or user ratings, we could envision a recommendation system based on user profiles and ratings to better select models. This seems to be a more long term feature to be considered.

(11) *As a researcher, I want to create communities, so that I can have a common space to manage my models.* This characteristic has been discussed also in depth among the authors and no consensus has been reached. The idea is to allow the creation of communities inside the repository that could allow to group people around a community. It could be the case of a research group or university that has its own space to share their models.

(12) *As a user, I want to see model's metrics, so that I can have extra information about the models.* There are some implicit information in feature models such as the number of features, the number of relationships or other structural metrics [1]. Having the possibility to obtain this information available in the repository would be of help for people using it. Some of the information can be computationally hard to calculate (such as number of dead features) so a trade-off among information availability and computation capacity has to be defined.

3 INFORMATION REQUIREMENTS

To implement the features described in Section 2, there is some information that would be useful to store among the model:

- File. The concrete file of the model. This is, the serialization of the model itself so it can be used across different platforms and tools (e.g. FaMiLiar, FaMa or Feature IDE)
- Author. The author of the model. Information of the creator of the model so it can be used to credit authors.
- Owner. The owner of the model that can be different of the author. It might happen that different users took the time to translate and transcribe the model from other languages or visual formats.
- Hash. An identifier of the model that can be used also as a checksum. This is useful for the integration within other tools so we can avoid wrong downloads.
- Description. A brief description of the model.
- Organisation/Community. In the case Communities are allowed, this information has to be stored.
- Language level. In the case different language levels are defined for the language, the model has to define at which level it belongs to. This is, if we ended up having a level supporting attributes, we would need to specify if the model contains such information.
- Rating. In the case models ratings are allowed. We can think of popularity of models and ratings from the community.
- Tags. To associate free tags to a model that can serve as a way to search and filter the models
- Version. In the case that different versions of the same model are allowed. This is intended to allow the upload of different versions of the same model. For example, showing how a model has evolved.
- Model's metrics. In the case that these metrics are allowed, this information has to be stored. To know in advance some characteristics of the models such as te number of features or cross-tree relationships ratio.

4 DEPENDENCIES WITH LANGUAGES ELEMENTS

There are several elements of the language that will affect the development of the repository. Note that these aspects should be discussing during the workshop:

4.1 Concrete syntax

The serialization for the selected feature model concepts can impact how we store the models. In the literature we can find multiples styles [4] of serializations for feature models. However, there are three main styles: *i*) XML based, multiple tools such as FaMa, SPLOT and Feature IDE rely in such representation. This representation has as main drawback the size of storage of each model and how hard to read visually can be. The main benefit is that we can rely on multiple available tools to read and write XML files; *ii*) Bracelet-based format: There are also tools that moved towards a more json alike formats (e.g. FaMiLiar) in which the hierarchical relationships are developed based on brackets. This ends up resulting in lower size which is beneficial for transmission and storage of the models. Those models are also more easy to read by a human. *iii*) Plain text Formats: These file formats have defined custom syntax to define each element of the feature model notation they work with. Some tools that provide support for this formats are FaMa and Feature IDE.

4.2 Abstract syntax/Model levels

In the literature we can find multiple variants of feature models that are capable of representing different facets of variability. Usually, two different groups of relationships are defined: *i*) hierarchical relationships to define the options enabled by a variation point in a product line and *ii*) cross-tree constraints to define restrictions on features that do not share a common parent in the feature model tree. We can define up to five flavours of feature models:

Basic feature models. These set of models are the most constrained ones and its first introduced in the FODA study [7] paper. They only encode Boolean features; four types of structural relationships. Namely, mandatory, optional, set and or. Finally, two types of cross-tree relationships requires and excludes. The basic feature model defines four kinds of hierarchical relationships: *i*) **mandatory**; *ii*) **optional**; *iii*) **set** which includes **alternative**; and **or** relationships; In addition to these hierarchical constraints, two cross-tree constraints are defined: *i*) **requires**, and; *ii*) **excludes**.

Cardinality based feature models. This representation is an attempt to unify feature models and UML constraint notations. This feature models definition, replaces the definition of basic feature model relationships by cardinality-based relationships.

Complex-constraints feature models. This representation extends previous levels by allowing the definition of any complex cross-tree constraint.

Attributed feature models. Attributed feature models is a feature modelling extension that includes non-Boolean information about features. Complex constraints are allowed between features and attributes such as "Every attribute called cost must have a value greater than 10." There are a variety of approaches to describe feature model attributes, however, most of them share some characteristics such as name and value.

Non-traditional feature models. This last group of feature models extends the previous approaches by adding functionality to encode more complex variability. For example, in this group we can find models containing multi-features [3], when a feature can be present more than one time in a product.

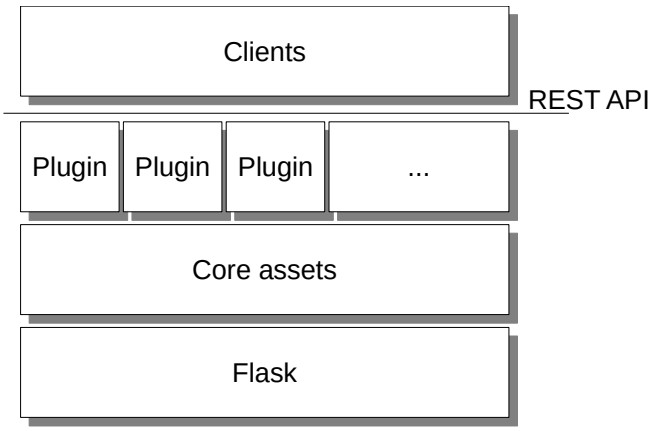


Figure 1: Possible architecture for our repository.

Our repository aims at supporting multiple flavours of the same model (e.g. with and without attributes) we would need to define the extension points of our language and the mechanisms to compose it before uploading it to the repository. That is, we would need to support versions of the same model but compatible with different levels of the language.

4.3 Technological stack to build the language infrastructure

We can summarise that the stack needs to fulfil the following requirements given the requirements shown in Section 2:

- Extensible. This repository feature responds to the need of providing rest interfaces, support for multiple attributes per model as well as the creation of communities.
- Light. Due the amount of optional features identified it would be interesting to rely on a small core.
- Reliable. There are several repositories that were built during the last years. It would be advisable that the core component of our repository is backed up by a larger project.

Figure 1 presents a possible architecture for our repository. First, we chose to select the micro framework Flask [2] which provides a light and extensive core for a web application and it is implemented in Python. This will enable the support of basic functionality such as accepting requests and build up our application. Second, a set of core assets providing the basic functionality of hosting and organising the models is needed. Then, we envision a plugin system to provide other functionality such as the evaluation of metrics for models or enable the rating of them. Finally, an API is provided to integrate the repository in different tools. Also, an HTML front-end will be built consuming such API.

Given those considerations, we propose to reuse and promote open-science previous efforts. Concretely, we found a solution stack

[2] http://flask.pocoo.org/

provided by Invenio [3] which was initially developed at CERN and holding an open-source licence.

5 CONCLUSIONS

We expect that the number of requirements would require some extensiveness in the repository so we can keep on adding new functionality as time goes by. For example, functionality such as the execution of analysis operations can be kept out of an initial version.

Also, this modular architecture would enable to adapt the repository for different levels of variability or serialisations. Another required main feature is the REST API so its easily interoperable with various tools and frameworks.

We think that the Invenio solution can serve as a base for building up a feature model repository that fulfills all our requirements. During the workshop, we aim to retrieve such information and reach consensus on the needs the community has.

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 Andalucia METAMORFOSIS project.

REFERENCES

[1] Ebrahim Bagheri and Dragan Gasevic. 2011. Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal* 19, 3 (2011), 579–612. https://doi.org/10.1007/s11219-010-9127-2

[2] Mike Cohn. 2004. *User stories applied: For agile software development.* Addison-Wesley Professional.

[3] 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 Press, 472–481.

[4] Holger Eichelberger and Klaus Schmid. [n. d.]. Textual Variability Modeling Languages: An Overview and Considerations. In *Proceedings of the 1st International Workshop on Languages for Modelling Variability (MODEVAR 2019).*

[5] Holger Eichelberger and Klaus Schmid. 2015. Mapping the design-space of textual variability modeling languages: a refined analysis. *STTT* 17, 5 (2015), 559–584. https://doi.org/10.1007/s10009-014-0362-x

[6] José A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated analysis of feature models: Quo vadis? *Computing* 101, 5 (01 May 2019), 387–433. https://doi.org/10.1007/s00607-018-0646-1

[7] Kang. 2010. FODA: Twenty Years of Perspective on Feature Modeling.. In *Proceedings of Fourth International Workshop on Variability Modelling of Software-Intensive Systems, ICB-Research Report, volume 37, Universität Duisburg-Essen, page 9.*

[8] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. SPLOT: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 761–762.

[9] Brian A Nosek, George Alter, George C Banks, Denny Borsboom, Sara D Bowman, Steven J Breckler, Stuart Buck, Christopher D Chambers, Gilbert Chin, Garret Christensen, et al. 2015. Promoting an open research culture. *Science* 348, 6242 (2015), 1422–1425.

[3] http://inveniosoftware.org/

Usage Scenarios for a Common Feature Modeling Language

Thorsten Berger
Chalmers | University of Gothenburg
Sweden

Philippe Collet
Université Côte d'Azur, CNRS, I3S
France

ABSTRACT

Feature models are recognized as a de facto standard for variability modeling. Presented almost three decades ago, dozens of different variations and extensions to the original feature-modeling notation have been proposed, together with hundreds of variability management techniques building upon feature models. Unfortunately, despite several attempts to establish a unified language, there is still no emerging consensus on a feature-modeling language that is both intuitive and simple, but also expressive enough to cover a range of important usage scenarios. There is not even a documented and commonly agreed set of such scenarios.

Following an initiative among product-line engineering researchers in September 2018, we present 14 usage scenarios together with examples and requirements detailing each scenario. The scenario descriptions are the result of a systematic process, where members of the initiative authored original descriptions, which received feedback via a survey, and which we then refined and extended based on the survey results, reviewers' comments, and our own expertise. We also report the relevance of supporting each usage scenario for the language, as perceived by the initiative's members, prioritizing each scenario. We present a roadmap to build and implement a first version of the envisaged common language.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

software product lines, feature models, unified language

ACM Reference Format:
Thorsten Berger and Philippe Collet. 2019. Usage Scenarios for a Common Feature Modeling Language. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3307630.3342403

1 INTRODUCTION

Feature models can arguably be seen as the most successful notation to model the common and variable characteristics of products in a software product line [11]. Proposed almost three decades ago, as part of the feature-oriented domain analyis (FODA) method [35], hundreds of variability management methods and tools have been built upon feature models. Of 91 variability management approaches introduced until 2011 [19], 33 have used feature models to specify variability information. The reported use of feature-modeling concepts in large commercial [9, 11] and open-source systems that have to manage different forms of variability, such as the Linux kernel [12, 13, 51] further witnesses their relevance. As an illustration, Fig. 1 shows an excerpt of the Linux kernel's model in a visual feature-modeling notation (explained in more detail in Sec. 2).

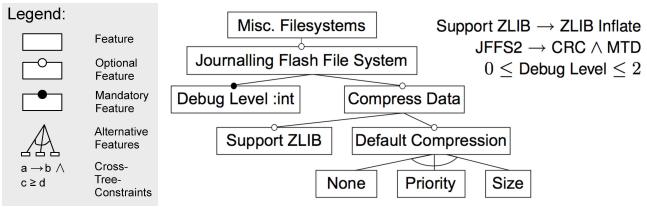


Figure 1: Feature model example (from [10])

Since the proposal of feature modeling in 1990, dozens of extensions and modifications have been proposed for feature models, often with the goal in mind to build a general feature modeling language, gradually extending the expressiveness of feature models. Examples are cardinality-based feature models [23], which support the multiple instantiation of features; attributed (a.k.a., extended) feature models [6], which allow features to have non-Boolean attributes (carrying, for instance, non-functional properties); more expressive (i.e., non-Boolean) constraint languages [46]; or even more radical approaches that combine feature and class modeling in one language [5]. Furthermore, while most academic feature-modeling notations are visual, many languages exhibiting a textual syntax have been developed for feature modeling [13, 14, 20, 27, 28].

Tooling or API-based support also emerged with the success of feature modeling in practice and research. The commercial product-line engineering tools pure::variants [17] and Gears [38], as well as the open-source tool FeatureIDE [55], are built upon feature models. In addition, many different feature-model analysis techniques and tools have been proposed [6, 42, 57]. Recent work also addressed the relative absence of processes for feature modeling by proposing modeling principles for engineers creating feature models [44].

However, despite this recognition of feature modeling in research and practice, there is still no emerging consensus on a language that would enable variability modeling in a simple and common way, while covering different possible usage scenarios. The attempt to build a standard for a common variability language, namely CVL [31], was dropped due to legal, patent-related issues. The language and its infrastructure is still available, however [58]. Establishing a standard would facilitate the interoperability of tools and would ease the sharing of feature models. Recognizing this pressuring need, a recent initiative among product-line researchers, driven by David Benavides, attempts to establish a common and simple, yet reasonably expressive feature-modeling language.

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342403

This paper summarizes one of the follow-up actions that were discussed at a meeting during the Software Product Line Conference (SPLC) in September 2018 in Gothenburg. There, after a brainstorming, a set of general usage scenarios of the prospective language was elicited. After a vote, 15 main scenarios of usage were extracted, and typically two researchers, one writing, another proofreading, were assigned to detail each scenario through a common template. Each scenario has a name, a small description, an example, and some additional notes with requirements or open questions related to it. These scenarios were then described during a one-month period from mid September to mid October 2018. A survey was then built to evaluate both the clarity and the usefulness (*i.e.*, priority) of each scenario. It was distributed on the mailing list created for the initiative at the end of October 2018. Analyzing the results, we refined and extended the scenario descriptions according to the results, as well as we removed and added a few scenarios.

2 BACKGROUND

We provide an introduction into feature-modeling concepts as well as a brief recap on the history of feature-modeling languages.

2.1 Feature Models

Figure 1 shows a small feature model describing the configurable filesystem JFFS (Journalling Flash File System) in the Linux kernel. Feature models are tree-like structures of features organized in a hierarchy together with constraints among the features. While the feature hierarchy is one of the most important benefits of feature models (called ontological semantics), allowing engineers to keep an overview understanding of a product line, the primary semantics (called configuration space semantics) of feature models is to represent the valid combinations and values of features in a concrete product of a product line, restricted by constraints as follows.

In our example, the feature Debug Level is a mandatory feature (filled circle) with the value type integer; Compress Data is an optional feature (hollow circle) of type Boolean with the optional sub-features Support ZLIB and Default Compression. The latter is a feature group of type XOR, allowing to select exactly one sub-feature. Other typical kinds of feature groups are OR and MUTEX groups (not shown). These constructs express constraints, in addition to the hierarchy constraints (a sub-feature always implies its parent feature). Further constraints, named cross-tree constraints, can be expressed separately to fully capture the configuration space—shown to the right of the diagram in our example (note that ZLIB Inflate is a feature that is defined outside our excerpt of the Linux kernel model, which has grown to around 15,000 features nowadays).

2.2 Feature Modeling Extensions

The original FODA feature models have been extended in many ways. Partly inspired by a genealogy of feature-modeling successors from Kang [34], the following major extensions have been proposed.

FORM feature models [36] were introduced as part of the feature-oriented reuse method (FORM) and sub-divided models into four layers, from abstract on top to very concrete implementation-oriented features at the bottom.

FeatuRSEB feature models [30] were introduced with the FeatuRSEB methodology, aiming at an integration with use case diagrams and similar models. They are mostly equivalent to FODA models.

Hein et al. feature models [32] introduced typed relationships and binding times for features, based on industrial experience that FODA "does not provide the necessary expressiveness to represent the different types of crosslinks" in their domain. Typed relationships give rise to alternative hierarchical structures in one model, so the diagram is a directed acyclic graph, not a tree anymore.

Generative Programming feature models [21] introduced the current notation and OR groups. This notation was later extended with typed attributes and feature cardinalities [25]. Furthermore, Riebisch et al. [47] introduced arbitrary group cardinalities and constraint notations. The most significant extension were feature cardinalities [23, 24], where features, and their whole subtrees, can have more than one instance in a configuration, which has considerable impact on reasoning operations and tooling.

Clafer [5, 33] is one of the most expressive feature-modeling languages, unifying feature and class modeling. The notion of feature and that of a class is unified into a Clafer, which has a name, types, constraints, and perhaps attributes. Clafer supports multi-level modeling [18] and has a well-sepcified semantics as well as rich tooling, including instance generation, configuration, and visualization. In addition, as a textual language, it has one of the simplest and most intuitive syntaxes. Developers can use a text editor and define a feature by writing its name into a line. Hierarchy is realized by indentation (similar to Haskell and Python). Making a feature optional amounts to adding a '?' character. Feature types can also be added in a simple way.

Kconfig and CDL [13, 14] are languages to describe the variability of systems. They are developed fully independently of the research community, by practitioners who were likely not aware of the existing feature-modeling languages from researchers. Kconfig and CDL are two of the most successful languages, primarily used in the systems software domain. Kconfig [51] is used in systems such as the Linux kernel, the Busybox project, and embedded libraries (e.g., uClibc). CDL [12] is used in eCos (embedded configurable operating system). The languages in fact use concepts known from feature modeling, including Boolean, int, and string features; a hierarchy; feature groups; and cross-tree constraints. However, the languages also bring additional concepts, mainly to scale feature modeling. Specifically, they provide: visibility conditions, modularization concepts, derived defaults / derived features, and hierarchy manipulation. In addition, they provide expressive constraint languages with three-state logics for binding modes, as well as comparison, arithmetic, and string operators. Finally, all use domain-specific vocabulary, enhancing their comprehension for the developers of the systems. Details are found in Berger et al. [13, 14]. Extracted models and an infrastructure to analyze them is also available online.[1]

In addition to extensions brought by these languages (e.g., diagram shapes, layers, binding modes, expressive constraints, cardinalities, and typed edges) we find some further concepts in the literature. Among these are defaults [21, 48] and visibility conditions [26]. The latter are usually part of decision modeling languages, which share many commonalities with feature models [22].

[1] https://bitbucket.org/tberger/variability-models

Table 1: Refined scenarios descriptions: modified slightly (○) or substantially (◐) by us; or is completely new (●).

	scenario	description	example	details
Christoph Seidl	**Exchange** ◐	The language should support the bidirectional exchange of feature models between different tools. Tool vendors use the language documentation and/or existing serializers/deserializers to create important and expert functionality. Users of the tools can then leverage this functionality to export a feature model from one tool and import it in the other tool.	A feature model is created in the source tool FeatureIDE [55] and then exported into a file with the concrete syntax of the language. The file can then be imported into the FAMA framework [57] for specialized feature-model analyses.	Requirements: • The language should have a serializable concrete syntax. • The language should come with sufficient documentation about its abstract and concrete syntax to realize importers and exporters. • Ideally, serializers and deserializers are provided for the language in the form of a library (in common programming languages, especially Java) that can be used by tool vendors. • The language may be extensible and an instance may describe the level of extensions that is used. • The language may provide concepts to store tool-specific data. • Storing tool-specific data should not require changing the language or provided serializers and deserializers. Open questions: • Should tool-specific data be kept in specific language concepts or should there be tool-independent concepts to store any kind of tool-specific data? Finding a middle ground might be necessary.
Thorsten Berger	**Storage** ●	The language should allow tools to efficiently store and load feature models. Tools can use the language and its concrete syntax as the primary means to store models. Tool vendors leverage the language specification to realize fast storage and loading of models. Two subscenarios are possible: (i) the model is stored in a database, and (ii) the model is stored in a textual representation.	Consider a new product line tool that needs to store feature models. The tool vendor can develop its persistence layer by creating leveraging the language specification (i.e., the abstract syntax definition) to derive a database schema and generate CRUD functionality as well as initialize the database.	Requirements: • The language should come with an abstract syntax definition in a metamodeling notation that can be used for automated processing (e.g., generate database schemas). • The language should come with a concise and succinct [49] textual syntax. • The textual syntax should be defined in a common technology for defining concrete syntaxes, such as an ANTLR or an Xtext grammar, both of which can be used for automated processing. Open questions: • Select a language workbench (e.g., Xtext [16], MPS [15], EMF [53]) or a parser-generator technology (e.g., ANTLR [45])?
Klaus Schmid, Rick Rabiser	**Teaching and learning** ◐	The language should be easily usable for teaching. Specifically, it should be possible to describe the language within a few slides, using concepts typically taught in computer science education (e.g., types, grammars, meta-modeling). Furthermore, the language's concepts should align well with the typical and established concepts (cf. Sec. 2) that have been introduced in the product-line community and are typically taught in SPL courses (features, attributes, constraints).	The teacher describes the language with fewer than a dozen slides, and the students are able to read and write simple examples afterwards.	Requirements: • The language should have the typical visual concrete syntax of feature models. • The language should come with realistic examples (ideally extracted from real-world models, such as the Linux kernel models [13], but toy models can also be provided for simplicity, such as from SPLOT [39]. • Ideally, the language also has a concrete textual notation to illustrate how to scale models. Open questions: • Teach the textual or graphical notation? • How to keep the language simple, while being expressive? • There is a need to understand the specific examples to be provided. • Should there be different levels to be taught? (corresponding to different levels in teaching) • When teaching, can we easily relate the key concepts of the language with standard concepts taught in computer science such as requirements, components, modules (e.g., "a feature can represent a requirement")
Rick Rabiser, Philippe Collet	**Writing, reading, and editing** ○	The language should support users in writing, reading, and editing feature models in a standard text editor, targeting developers or modelers with basic programming language knowlege. Tool vendors should be able to use the language specification with automated tooling to generate an infrastructure for using the language, with typical software language engineering or transformation technology (e.g., XText, XTend or Coco/R [41]).	The user opens an editor and, given some basic knowledge about the key constructs of the language, she can instantly start writing feature models. A domain expert can easily edit feature models inside the same kind of editor. The generated language infrastructure contains a modern editor with syntax formatting, highlighting, code completion, and syntax checking.	Requirements: • The language should provide a simple and human-readable textual concrete syntax. • The language definition should be independent of a particular generation technology. • The language should allow the use of standard text editors. • Instances should be editable in standard IDEs, such as Eclipse, IntelliJ IDEA or Microsoft Visual Studio. • The language's parser should be easy to integrate in other tool chains. • Ideally, the requirements of the scenario **Domain modeling** apply.
David Benavides, José Galindo	**Model generation** ◐	Model generation (a.k.a., instance generation) automatically creates instances (models) of the language, typically aiming at instances with certain properties, such as size, coverage of language concepts, or other structural characteristics (e.g., cross-tree constraints ratio [8, 40, 50]). Tool developers can use it to generate a set of models, useful for functional testing and performance testing of the different tools supporting the language.	A tool developer launches the instance generation tool, inputs the desired properties of the model to be generated, and obtains the desired model(s).	Requirements: • The language specification (syntax and semantics) should allow for a translation of the complete semantics into a representation in a formal language. • The formal language should allow instance generation (e.g., Alloy), with instances that can be expressed in the original language's syntax (so, instantiated model in the formal lanugage should be structurally similar to the target model in the new feature-modeling language). • Ideally, the instance generation can be interactive, also showing conflicting constraints and counter-examples.

	scenario	description	example	details
Thorsten Berger	**Domain modeling** ○	The language should support early and creative software-engineering phases by allowing concept/domain modeling in terms of features. Specifically, it should allow creating features in a hierarchy, without having to specify feature value types, feature kinds (mandatory, optional), feature cross-tree relationships, or whether they belong to a feature group. The model can be gradually refined with those concepts later. Furthermore, if typed relationships are supported, hard and soft (e.g., recommends) constraints could be distinguished, the latter can be defined for each model.	A user creates an empty model and in parallel adds features (that are simply characterized by their names) and organizes them in a hierarchy. She adds cross-tree relationships if she finds it useful and quickly re-organizes the hierarchy when the domain model becomes more clear. Later, when the structure is more stable, she defines which features are mandatory, which optional, as well as she defines the other concepts.	Requirements: • The language should provide a simple and human-readable textual concrete syntax. • The language should have a concise and succinct textual syntax. • The language should rely on conventions and defaults that allow omitting the explicit instantiation of concepts (e.g., when not specific, the default feature type should be Boolean). • The textual syntax could be inspired by Clafer (cf. Sec. 2). Open questions: • Domain/concept modeling might require: multiple feature instantiation (cardinality-based feature modeling, cf. Sec. 2) as well as multi-level modeling and ontological instantiation [18]. However, supporting these concepts (as is supported by Clafer), could complicate the language. • Support typed relationships?
Christoph Seidl, Klaus Schmid	**Configuration** ◐	The language should support the configuration activities of a feature model. It should include respective constructs for selection, de-selection, un-selection of features in an feature model under configuration. Resolution of the resulting configuration space after such configuration operations should be supported by the language. The language should also support partial configuration management. The language should support default values as in product configurators. This could be directly supported in the feature model (e.g., for alternatives).	A previously created feature model is used to determine functionality of a particular variant by selecting individual features. A partial configuration of a feature model is done by several selections and unselections. The resulting set of configurations is available in the language. The absence of any resulting configuration is detected after a conflicting set of selections (e.g., with a cross-tree constraint being violated).	Requirements: • Provide adequate syntax for configuration by non-technical stakeholders. • A configuration comprises selected features but, with more elaborate language constructs (e.g., attributes), should also include value selection. • Default configurations or exemplary configurations may be sensible as suggestions (e.g., "portfolios/profiles"). • Support partial configuration. • Support by inference engine requires different types of constraints: those that can be violated temporarily and those that cannot—a typical distinction in practical languages used for configuration. Open questions: • Should the configurations be persisted within a feature model or external to it? • Is there a unique name assumption so that features can be referenced unambiguously by name? • Are configurations first-class entities in the language? • In the case of a partial configuration, should the resulting (refined) feature model be available in the language together with the set of possible configurations?
David Benavides, Mathieu Acher	**Benchmarking** ◐	The language should be designed for tool support, and several implementations are expected to be available. There should be a well-defined set of indicators to measure the performance of the most relevant operations (e.g., analysis, refactoring, configuration completion), so to be able to compare them. The benchmarking setup would allow to compare tool support execution times of these operations in isolation (e.g., without taking into account file loading or feature model parsing times when focusing on a reasoning operation).	The user loads the model with FAMA [57], Familiar [4] or Feature IDE [55] and executes the operation 'dead features,' also measuring the completion times. Then she knows which is the best tool for that operation and model. Each tool built upon the language can run the common benchmark and automatically produce an exploitable performance result.	Requirements: • Well-engineered and specified syntax and semantics of the language. • There should be an agreement on the specificatin of certain feature-model operations. • The availability of realistic models is important. Potentially, real-world models from the systems software domain can be used (cf. Sec. 2)
José Galindo	**Testing** ◐	Feature models expressed in the language should be usable as input for testing, specifically, for configuration sampling. Features and especially their constraints should be extractable in a form that allows reducing the search space for sampling techniques. Another strategy to support testing would be to express full or partial configurations (e.g., pairs of features that are critical to test).	A software engineer creates test cases that will run with configurations that are recorded in the feature model. Furthermore, when an unwanted feature interaction is detected, the engineer will record the feature pair, to be considered in future regression testing.	Requirements: • Language concepts to represent partial or complete configurations. • Ideally, consistency checking by the language infrastructure for the configuration information. • Incorporating concepts capturing further testing-relevant information (e.g., inputs for testing dedicated features) could be useful. Open questions: • Should testing-related configuration information be stored directly in the model or in a separate kind of asset?
David Benavides, Philippe Collet	**Analyses** ◐	The language can be used in automated analysis processes where the model is used as input and an analysis result is obtained. This can comprise analyses confined to the feature model [6, 29] or those that take other artifacts into account [42, 54].	Consider a Linux distribution, such as Debian. Let us assume the packages (each representing a feature) and their dependencies are described using our language (or, more realistically, are transformed from Debian's manifests into a feature model). An off-the shelf analysis, such as "dead features" can then be used to detect packages that are not selectable.	Requirements: • Community agreement on a core set (or class) of relevant analyses. • Consider different solver strategies depending on the kinds of analyses and the constructs of the language. For instance, if we allow attributes, then, specific solver capabilities are needed. • Well-specified language syntax and semantics, also with semantic abstractions into the different logical representations required by the solvers. Open questions: • Is the representation of correspondence (and maybe performance) of the solving strategies to the different constructs and extensions part of the language definition? • Should analyses be confined to the feature model or also take other asset types into account, such as the mapping to implementation assets?

scenario	description	example	details
Thomas Thüm	**Mapping to implementation** ○	Feature models are often not only considered in isolation. Instead, features are typically mapped to certain assets. Depending on the use case, features are mapped to requirements, architecture, design, models, source code, tests, and documentation, among others. While the actual mapping is largely independent of the feature modeling language, it should be possible to distinguish features that are supposed to be mapped to artifacts from those purely used to structure the hierarchy (e.g., to group certain features into an alternative group) or features that are not yet implemented. So, the scenario is to support developers mapping features to the implementation.	Suppose we implement a product line incrementally. That is, we have done a domain analysis in which we created a feature model and now we implement more and more of those features over time. Assume we want to derive a product or count the number of possible products before we are done with the implementation of all features. During configuration, we do not want to make decisions that do not influence the actual product. For counting, we are not interested in the total number of valid configurations, but only in those that result in distinct products.	Requirements: • A single modifier/keyword to be assigned to every feature could be sufficient (e.g., abstract/concrete as in FeatureIDE) • A well-defined mapping language might be necessary. • Avoid common limitations. For instance, a simple language rule as applied in GUIDSL, such that every feature without child features in concrete and all others are abstract, would result in unintuitive editors and overly complex feature models if a feature with child features is supposed to be mapped to artifacts. • A challenge is that this property is not be supported in many tools. Fallback could always be to mark all features as concrete during import/export (could be default). Open questions: • Should the mapping be part of the language or realized in a separate one?
Thomas Thüm	**Decomposition and composition** ◐	Industrial models tend to have thousands of features. Clearly, those models are created by numerous stakeholders that may even originate from different divisions or even institutions. Models can also be built according to specific separated concerns. The language should be able to compose several feature models according to different semantics (e.g., aggregation, configuration merging). It is often necessary to decompose such a large model into smaller pieces to improve the overview and facilitate collaborative development.	The Linux kernel is defined in the Kconfig [51] language but not in a single file. The knowledge is distributed over several files according to the structure of the code base. For analyses it is typically necessary to compose them all prior to feeding them into solvers. The largest known feature model (Automotive2 [37]) was developed in terms of 40 small models that have even used different modeling languages.	Requirements: • Prioritized list of composition mechanisms from the literature (e.g., aggregation, inheritance, superimposition, configuration merging). • Simple mechanism that is easy to implement. • Perhaps dedicated support for interface feature models (cf. principle MO_3 among common feature modeling principles [44]) Open questions: • Should all the composition mechanisms that have been discussed in the literature be supported in the language? • How should the language handle the fact that depending on the composition operator, it could be possible or not to add the same model several times within another model? • Is it sufficient to have support for composition in the language, whereas decomposition is up to the users?
	Model weaving ◐	The language should be able to be easily integrated with other programming languages for supporting variability modeling at design and implementation levels. Several integration levels could be considered depending on the host language's capabilities and engineering needed to provide such integration in the language. A shallow form of integration could be a simple interpreter available in the host language, exchanging input and output as strings and basic types. A deeper form of integration could be an API enabling to manipulate the high-level concepts of the language in the host language (e.g., features, feature models, configurations).	Currently there are different strategies for imposing variability in other modeling formalisms such as business processes or object-oriented design. An ideal scenario would be to use our language to integrate variability in other languages, such as BPMN.	Requirements: • Depending on the depth of the integration, static design-time or dynamic run-time model weaving might need to be considered. • List of variability mechanisms from the literature. • Understanding of the effort for realizing the mechanism for different types of assets. Open questions: • Separation of concerns is an issue to consider (support developers to separate problem and solution space). • The necessary extent of embedding of information from the feature model into other assets needs to be investigated, making realistic assumptions about the actual need.
Mathieu Acher, Tewfik Ziadi	**Reverse engineering and composition** ●	It should be possible to use the language to represent reverse-engineered or composed feature models. The former can originate from typical reverse-engineering techniques that rely on extracted variability information [2, 3, 43, 52], the latter can originate from multiple existing feature models.	Reverse-engineering examples: Synthesize a feature model from command-line parameters and the respective source-code in which they are used, from configuration files or from product comparison matrices; re-engineer web configurators [1]; extraction of feature models and reusable assets from clone & own-based systems. Composition examples: Combine multiple product lines; or use of composition and slicing operations over feature models to build a specific viewpoint on the variability.	Requirements: • The language should be sufficiently expressive to model real-world variability and configuration spaces that are reverse-engineered (these often have non-Boolean and complex constraints). • Perhaps some traceability (e.g., the artifacts from which certain constraints stem from) or debugging information (e.g., how the constraint was calculated or whether it is abstracted by weakening a constraint to make it processable). Open questions: • How to deal with constraints among features or other information extracted that is not expressible in the language (e.g., child features that exclude their parents in the Linux kernel model [13]. • As such, ground truth models may not be expressible in our language. • Empirical validation of the language: How to validate that our language is sufficient for reverse-engineering variability, especially from legacy systems? • Could round-trip engineering (which is a hard problem) be supported?

3 EMERGING USAGE SCENARIOS

As explained above (Sec. 1), the participants of the initiative's first meeting during SPLC 2018 in Gothenburg authored an original set of usage scenarios, each of which having a name, a description, an example, requirements, and potentially open questions for discussion. These were then evaluated in an online survey, created by David Benavides, targeting participants of the initiative—also those who did not attend the Gothenburg meeting, but are registered on the respective mailing list (featuremodellanguage@listas.us.es).

The survey elicited the perceived clarity of the scenario and the perceived usefulness. The original formulations of the scenarios are available in our online appendix [7], together with more detailed survey data. Specifically, using Likert-scale questions, for each scenario it asked:

(1) *What is the clarity of the scenario?* Either: 1 (not clear at all), 2 (not clear), 3 (more or less), 4 (clear), 5 (very clear).
(2) *What is the usefulness/priority of the scenario?* Either: 1 (not useful at all), 2 (not useful), 3 (more or less), 4 (useful), 5 (very useful).

The survey was distributed among the mailing list of 25 interested persons in the building of the common language at the end of October 2018 for a period of 7 weeks. 15 answers were collected.

3.1 Survey Analysis and Scenario Refinement

The results of the survey corresponded to 14 to 15 answers on each scenario, as one participant did not evaluate all scenarios. With respect to the clarity of each scenario, the results were very diverse, with some scenarios being mainly viewed as clear, while some others were not considered as clear enough. Consequently, we decided to improve the descriptions, as mentioned above (Sec. 1), to provide a better set of descriptions. Table 1 shows our final set of scenarios, refined and extended compared to those formulated after the initiative's first meeting. The table acknowledges the original authors, but we indicate whether we have modified the scenario only slightly (○), whether we did substantial changes (◐), or whether the scenario is completely anew (●). For the latter, the stated authors are also those of the new formulation.

Specifically, compared to the original set (cf. appendix [7]), we removed the scenarios **Language-Specific Characteristics** (since it was a design recommendation instead of a usage scenario or a requirement), **Storage** (since the text primarily described the sharing of models, which is covered by scenario **Exchange**), and **Translation to logics** (since it is not a prime usage scenario performed by a language user or tool developer as such, but represents a design decision and technicalities necessary to realize the majority of the other scenarios). We added the scenario **Reverse engineering and composition**, which was not formulated out by the time of the survey.

Figure 2 shows violin plots about the survey's answers with respect to the usefulness, indicating the scenario's relevance. A similar violin plot for clarity is available in our appendix [7]. Given the change, the result for **Storage** should be taken with care.

3.2 Design Recommendations

To some extent, the original usage scenario descriptions contained information about the realization of the language, which was out of the scope of this early phase of collecting usage scenarios for

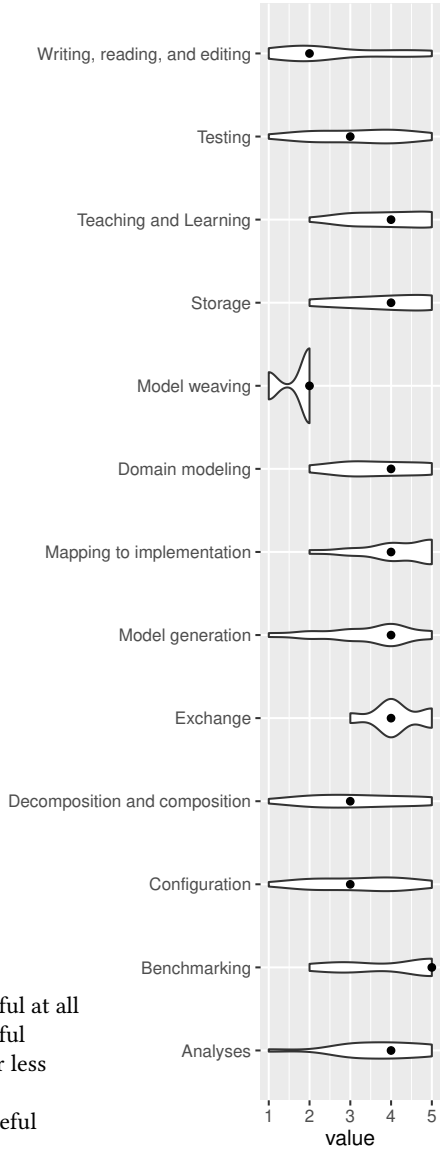


Figure 2: Perceived usefulness of the scenarios (without scenario reverse engineering and composition, cf. Sec. 3.1)

users and tool developers. The removed scenario **Translation to logics** is such a case, which provides valuable implementation recommendations by Thomas Thüm and Maurice ter Beek, further detailed in a separate paper [56]. Specifically, they argue that a key enabling technology for feature modeling is the ability to translate the semantics of feature models to a logical representation that can serve as an input to off-the-shelf constraint solvers. To this end, the expressiveness of the language should ideally align with relevant solvers, such as SAT, BDD, SMT or CSP solvers. To remain flexible, the language could offer different levels that classify the language concepts into different levels of expressiveness, each of which aligning with a specific class of solvers. Specifically, levels representing higher expressiveness allow more language concepts, but limit which solvers are applicable. For instance, if the language would

have a level that allows specifying non-Boolean feature attributes (representing, for instance, cost or performance properties) and quantitative constraints among them, this would exclude the use of solvers relying on propositional logics (e.g., SAT and BDD solvers).

To obtain language levels, Thüm and ter Beek propose mapping language concepts to levels of expressiveness, followed by identifying the priority of supporting each level based on the usage scenarios. For instance, a language level that allows quantitative modeling could support modeling (at least parts) of continuous behavior in cyber-physical systems. Thüm and ter Beek also note that language concepts might interact with respect to expressiveness, which needs to be taken into account when designing the language.

4 A PRELIMINARY ROADMAP

Our results suggest the following aspects to be considered in specifying the future language and in the workshop.

As we believe that we need to incrementally build the language features to make progress, a first set of features can be devised from the scenarios that are perceived primarily useful. According to the median value of the distribution shown in Fig. 2, this set would then comprise the following scenarios: **Exchange**, **Storage**, **Domain Modeling**, **Teaching and Learning**, **Mapping to implementation**, **Model generation**, **Benchmarking**, and **Analyses**.

From these scenarios, we can imagine some design decisions to be discussed and validated to get to a first version of the language:

- A simple textual language seems to meet the challenges from the scenario **Exchange**.
- Realization of the language's abstract and concrete syntax using a common language workbench (e.g., Eclipse EMF with Xtext) can support the scenario **Storage**.
- Incremental and partial creation of a feature model is needed for **Domain Modeling**. It directly affects the scope of what we could put inside the first set of functionalities.
- For a first set of functionalities meeting **Teaching and Learning**, simplicity of the language for writing, editing, and configuring, should be kept in mind.
- The scenarios **Model generation**, **Benchmarking**, and **Analyses** could be easy to meet in a first version if propositional feature models are chosen as a first level of expressiveness.
- *Mapping to implementation* is not an easy scenario to meet, as it is an open problem depending on the artifacts and variability realization technique.

Considering the current set of feature-modeling languages that are available, Clafer appears to meet most of the requirements. As described in Sec. 2, it is one of the most expressive languages while having a concise and succinct textual syntax, accompanied with formally defined semantics, and coming with substantial tooling. On the negative side, however, is the complexity and richness of the language, which might be problematic, but could be addressed by specifying a subset language level to accommodate certain subsets of usage scenarios.

With these aspects in mind, some open questions arise, as a basis for discussion during the workshop:

- Would the first kernel of functionalities of the language be designed and implemented at the same time? Implementation would enable to validate scenarios automatically through a continuous integration pipeline.
- Once the implementation subject is raised, the textual language implementation choices are raising at well: it could be a fluent API, an external or internal DSL, or a clever combination.
- Could the scenario **Analyses** be used with its first example, *i.e.*, running a dead-feature analysis, as a validation scenario for the implementation part of the language? Still, what analyses are useful and how their scenarios should be made clearer must be discussed. Similarly, could the **Benchmarking** scenario be also added in the same way, as its first example is a benchmark over the dead-feature computation?
- Could the language Clafer provide a reasonable basis for realizing the desired language, potentially by introducing language levels into Clafer, reducing its complexity for many scenarios?

For the workshop, we suggest to conduct a second evaluation of the refined and extended usage scenarios we presented in this paper. We plan to update our appendix [7] with the new results. The survey should again elicit clarity and usefulness, to increase our confidence in the scenarios. It should also re-open the discussion about further scenarios that need to be realized. For instance, a scenario that was briefly discussed during the first workshop was the collaborative creation of feature models, but not formulated out. Collaborative creation of feature models might be relevant for domain modeling; however, common wisdom on the processes and organizational aspects of feature modeling suggests that the distribution of the brittle variability information, and the maintenance of feature models by more than a small core group, is not feasible [10].

5 CONCLUSION

In this paper we contributed 14 usage scenarios for a simple and common feature-modeling language, to be finally established as a standard for feature modeling. We refined and extended formulations for a set of scenarios originally formulated by members of the initiative—experienced researchers from the product-line community that have some expertise in several facets of the creation and maintenance of important functionalities of such a language. We relied on a survey that was created for eliciting the clarity and relevance (i.e., usefulness or priority) of each scenario. We reported the survey results, presented the scenarios, and proposed a roadmap to support the next steps of the initiative. From these results, we observed the emergence of a smaller set of scenarios, seen as clearer and most useful, which could make a first kernel of the targeted language. We expect these insights to help in driving discussions and making decisions during the workshop.

ACKNOWLEDGMENTS

We would especially like to thank David Benavides for starting and steering this initiative, and for creating the survey questionnaire upon the initial scenario descriptions. We also thank the scenario authors Mathieu Acher, Maurice Ter Beek, David Benavides, José A. Galindo, Rick Rabiser, Klaus Schmid, Thomas Thüm, and Tewfik Ziadi; as well as we thank all the other participants of the initiative's first meeting in Gothenburg 2018.

Thorsten Berger's work is supported by Vinnova Sweden (2016-02804) and the Swedish Research Council (257822902).

REFERENCES

[1] Ebrahim Abbasi, Arnaud Hubaux, Mathieu Acher, Quentin Boucher, and Patrick Heymans. 2012. *What's in a web configurator? empirical results from 111 cases.* Technical Report P-CS-TR CONF-000001. University of Namur.

[2] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. 2011. Reverse Engineering Architectural Feature Models. In *ECSA*.

[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*.

[4] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. FAMILIAR: A Domain-specific Language for Large Scale Management of Feature Models. *Sci. Comput. Program.* 78, 6 (June 2013), 657–681.

[5] Kacper Bąk, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Feature and meta-models in Clafer: mixed, specialized, and coupled. In *SLE*.

[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] Thorsten Berger and Philippe Collet. 2019. Survey Data on Usage Scenarios for a Common Feature Modeling Language. Technical Note. Available at http://www.cse.chalmers.se/~bergert/paper/2019-tn-fml_survey.pdf.

[8] Thorsten Berger and Jianmei Guo. 2014. Towards System Analysis with Variability Model Metrics. In *VaMoS*.

[9] 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 *MODELS*.

[10] Thorsten Berger, Rolf-Helge Pfeiffer, Reinhard Tartler, Steffen Dienst, Krzysztof Czarnecki, Andrzej Wasowski, and Steven She. 2014. Variability Mechanisms in Software Ecosystems. *Information and Software Technology* 56, 11 (2014), 1520–1535.

[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 *VaMoS*.

[12] Thorsten Berger and Steven She. 2010. Formal Semantics of the CDL Language. Technical Note. http://www.cse.chalmers.se/~bergert/paper/cdl_semantics.pdf.

[13] 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. on Software Engineering* 39, 12 (2013), 1611–1640.

[14] 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 *ASE*.

[15] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of Projectional Editing: A Controlled Experiment. In *FSE*.

[16] Lorenzo Bettini. 2013. *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt.

[17] Danilo Beuche. 2004. pure::variants Eclipse Plugin. (2004). User Guide. pure-systems GmbH. Available from http://web.pure-systems.com/fileadmin/downloads/pv_userguide.pdf.

[18] Victorio A. Carvalho and João Paulo A. Almeida. 2016. Toward a well-founded theory for multi-level conceptual modeling. *Software & Systems Modeling* (30 Jun 2016).

[19] 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.

[20] Andreas Classen, Quentin Boucher, and Patrick Heymans. 2011. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming* 76, 12 (2011), 1130 – 1143.

[21] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, Boston, MA.

[22] K. Czarnecki, P. Gruenbacher, R. Rabiser, K. Schmid, and A. Wasowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *VaMoS*.

[23] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process Improvement and Practice* 10, 1 (2005).

[24] K. Czarnecki and C.H.P. Kim. 2005. Cardinality-based feature modeling and constraints: A progress report. In *International Workshop on Software Factories*.

[25] Krzysztof Czarnecki, Kasper Østerbye, and Markus Völter. 2002. Generative Programming. In *ECOOP*.

[26] Deepak Dhungana, Patrick Heymans, and Rick Rabiser. 2010. A Formal Semantics for Decision-oriented Variability Modeling with DOPLER. In *VaMoS*.

[27] Holger Eichelberger and Klaus Schmid. 2013. A Systematic Analysis of Textual Variability Modeling Languages. In *SPLC*.

[28] Holger Eichelberger and Klaus Schmid. 2015. Mapping the Design-space of Textual Variability Modeling Languages: A Refined Analysis. *Int. J. Softw. Tools Technol. Transf.* 17, 5 (Oct. 2015), 559–584.

[29] José A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated Analysis of Feature Models: Quo Vadis? *Computing* 101, 5 (May 2019), 387–433.

[30] M. L. Griss, J. Favaro, and M. d' Alessandro. 1998. Integrating Feature Modeling with the RSEB. In *ICSR*.

[31] Oystein Haugen, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. CVL: common variability language. In *SPLC*.

[32] Andreas Hein, Michael Schlick, and Renato Vinga-Martins. 2000. Applying Feature Models in Industrial Settings. In *SPLC*.

[33] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. 2018. Clafer: Lightweight Modeling of Structure and Behaviour. *The Art, Science, and Engineering of Programming Journal* 3 (07/2018 2018).

[34] K.C. Kang. 2009. FODA: Twenty Years of Perspective on Feature Models. In *Keynote Address at the 13th International Software Product Line Conference (SPLC'09).*

[35] 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.

[36] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. 1998. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.* 5 (Jan. 1998), 143–168.

[37] Sebastian Krieter. 2015. *Efficient Configuration of Large-Scale Feature Models Using Extended Implication Graphs.* Ph.D. Dissertation. University of Magdeburg.

[38] Charles W. Krueger. 2007. BigLever software gears and the 3-tiered SPL methodology. In *OOPSLA*.

[39] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: software product lines online tools. In *OOPSLA*.

[40] Marcilio Mendonça, Andrzej Wasowski, and Krzysztof Czarnecki. 2009. SAT-based Analysis of Feature Models is Easy. In *SPLC*.

[41] H. Mossenbock. 1990. *Coco/R - A Generator for Fast Compiler Front Ends.* Technical Report. Johannes Kepler Universität Linz.

[42] 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 *International Conference on Automated Software Engineering (ASE).*

[43] 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.

[44] Damir Nešić, Jacob Krüger, Stefan Stănciulescu, and Thorsten Berger. 2019. Principles of Feature Modeling. In *ESEC/FSE*.

[45] Terence Parr. 2013. *The definitive ANTLR 4 reference.* Pragmatic Bookshelf.

[46] Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. 2011. A Study of Non-Boolean Constraints in a Variability Model of an Embedded Operating System. In *FOSD*.

[47] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. 2002. Extending Feature Diagrams with UML Multiplicities. In *IDPT*.

[48] Juha Savolainen, Jan Bosch, Juha Kuusela, and Tomi Männistö. 2009. Default values for improved product line management. In *SPLC*.

[49] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *RE*.

[50] 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*.

[51] Steven She and Thorsten Berger. 2010. Formal Semantics of the Kconfig Language. Technical Note. http://www.cse.chalmers.se/~bergert/paper/kconfig_semantics.pdf.

[52] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse engineering feature models. In *ICSE*.

[53] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional.

[54] 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), 6:1–6:45.

[55] 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.

[56] Thomas Thüm, Christoph Seidl, and Ina Schaefer. 2019. On Language Levels for Feature Modeling Notations. In *MODEVAR*.

[57] Pablo Trinidad, David Benavides, Antonio Ruiz-Cortés, Sergio Segura, and Alberto Jimenez. 2008. Fama framework. In *SPLC*.

[58] Anatoly Vasilevskiy, Øystein Haugen, Franck Chauvel, Martin Fagereng Johansen, and Daisuke Shimbara. 2015. The BVR tool bundle to support product line engineering. In *SPLC*.

Should Future Variability Modeling Languages Express Constraints in OCL?

Don Batory
University of Texas at Austin
batory@cs.utexas.edu

ABSTRACT

Since the mid-2000s, *Propositional Logic* (PL) has been the de facto language to express constraints in *Feature Models* (FMs) of *Software Product Line* (SPLs). PL was adequate because product configurations were formed by binary decisions including or not including features in a product. Inspired by both prior research and practical systems (*eg.*, SPLs that use KConfig), future FMs must go beyond PL and admit numerical (and maybe even text) variables and their constraints.

The *Object Constraint Language* (OCL) is a general-purpose declarative constraint language for *Model Driven Engineering* (MDE), which admits virtually any kind of variable and constraint in metamodels. We should expect future FMs to be examples of MDE metamodels. This raises a basic question: Should OCL be used to express constraints of future variability modeling language(s)?

In this talk, I outline the pros and cons for doing so.

ACM Reference Format:
Don Batory. 2019. Should Future Variability Modeling Languages Express Constraints in OCL?. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 1 page. https://doi.org/10.1145/3307630.3342406

1 INTRODUCTION

I am no fan of the *Object Constraint Language* (OCL) and never have been. I find it inelegant and bloated. Using Eclipse OCL years ago, I recall different OCL implementations did not agree on syntax and covered the OCL standard (at that time) with varying fidelity. (Note: The current 2.0 standard is a behemoth 262 page document [5]!) I tried to teach OCL to my undergrads, and that was an unpleasant experience for both me and my students. OCL and related tools were simply too complicated. As mentioned in [1], Eclipse tools:

(1) Were unappealing—they were difficult to use even for simple applications.
(2) Fostered a medieval mentality in students to use incantations to solve problems. Point here, click that, something happens. From a student's perspective, this is gibberish. Although I could tell them what was happening, this mode of interaction left a vacuum where a deep understanding should reside.
(3) Have a steep entry cost to use, teach, and learn – too high for my comfort (and I suspect my student's as well).

I'm not alone with these opinions [2].

From a distance, I have also watched various attempts to generalize *Feature Models* (FMs) to address next generation *Software Product Line* (SPL) concerns – such as admitting replicated features, features with attributes, numerical features, and expressing constraints. I recoiled at the complexity of these attempts, and the use of OCL as the language to express constraints.

I do not profess to know what future SPL Variability Modeling Languages will be and how constraints in such languages will be expressed. But I do believe the answer will be guided by:

- **Simplicity!** PL was chosen for classical FM constraints because it was a simple mathematical standard. I'm not sure there is a formal grammar (language) to which all classical FM tools agree, but it is hard to screw-up writing PL constraints. Next-generation FM constraints should be equally straightforward to write.
- **Don't Invent, Reuse!** Do we really need a **new** constraint language for future FMs? Clearly we need more than PL. But are we good enough as language engineers to create a new constraint language without making a complete mess of it? Shouldn't we reuse existing languages or sub-languages of existing languages? Our expertise is in SPLs, **not** in language engineering. If you want an example of (IMO) a failed custom constraint language, it is OCL. Re-read the 2nd sentence of this Introduction.
- **Circularity Avoidance!** Generalizing beyond hierarchical relationships of classical FMs, we're not far away from UML class diagrams [4] and Model Driven Engineering *metamodels* [3] – which are class diagrams + constraints. And constraints for class diagrams beg the use of OCL.

In this talk, I offer and demonstrate a way out of this circular conundrum. My solution does *not* eliminate all problems, but it does diminish key problems about OCL standards, OCL tooling, reducing the need for yet-another-language, minimizing long-term tool maintenance, and keeping constraint languages both familiar and simple to SPL programmers, practitioners, and researchers.

Acknowledgments. Batory is supported by NSF grant CCF1212683.

REFERENCES

[1] D. Batory and M. Azanza. Teaching model-driven engineering from a relational database perspective. *Softw. Syst. Model.*, May 2017.
[2] J. Cabot and M. Gogolla. Object constraint language: A definitive guide. https://www.slideshare.net/jcabot/ocl-tutorial.
[3] K. Czarnecki, Chang Hwan, P. Kim, and K. T. Kalleberg. Feature models are views on ontologies. In *SPLC*, 2006.
[4] M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley Longman Ltd., 1997.
[5] About the object constraint language specification version 2.4. https://www.omg.org/spec/OCL/About-OCL/, 2019.

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342406

An Industrial Case Study for Adopting Software Product Lines in Automotive Industry

An Evolution-Based Approach for Software Product Lines (EVOA-SPL)

Karam Ignaim **João M. Fernandes**
Department of Informatics / ALGORITMI Centre
Universidade do Minho
Braga, Portugal

ABSTRACT

Software Product Lines (SPLs) seek to achieve gains in productivity and time to market. Many companies in several domains are constantly adopting SPLs. Dealing with SPLs begin after companies find themselves with successful variants of a product in a particular domain. The adoption of an SPL-based approach in the automotive industry may provide a significant return on investment. To switch to an SPL-based approach, practitioners lack a reengineering approach that supports SPL migration and evolution in a systematic fashion.

This paper presents a practical evolution-based approach to migrate and evolve a set of variants of a given product into an SPL and describes a case study from the automotive domain. The case study considers the need to handle the classical sensor variants family (CSVF) at Bosch Company.

Using this study, we performed a contributed step toward future switch of the CSVF into the SPL. We investigated the applicability of the proposed evolution-based approach with a real variants family (using the textual requirements of the CSVF) and we evaluated our approach using several data collection methods. The results reveal that our approach can be suitable for the automotive domain in the case study.

CCS CONCEPTS

• Software and its engineering → Software development techniques → Reusability

KEYWORDS

Software Product Line, Feature Model, case study, variability.

ACM Reference format:

Karam Ignaim and João M. Fernandes. 2019. An Industrial Case Study for Adopting Software Product Line in Automotive Industry: An Evolution-based Approach for Software Product Lines (EVOA-SPL). In *Proceedings of ACM 23rd International Systems and Software Product Line Conference (SPLC 2019). Paris, France, 8 pages.*
https://doi.org/10.1145/3307630.3342409

SPLC '19, September 9–13, 2019, Paris, France.
 ACM ISBN 978-1-4503-6668-7/19/09…$15.00. https://doi.org/10.1145/3307630.3342409

1 Introduction

Some companies in the market need to handle multiple variants that have some characteristics in common [1]. One of the key factors to improve productivity and consequently to reduce the cost to handle multiple variants is software reuse. In fact, many variants in a specific industrial domain have been implemented by reusing pieces of existing variants rather than building them from scratch [2]. An SPL can be seen as a family of variants that have been developed with explicit concern about commonality and variability during development process [3].

SPLs have been used successfully by industry to promote reuse. For example, several reports from large companies such as Bosch, Nokia, and Philips observe benefits with their use, especially with respect to the reduction in time to market [4]. Because of long term dealing with variants family, companies need to handle them in a systematic way. Therefore, SPLs can be a suitable option in this case. To switch to SPLs, companies need to adopt an approach that supports migration of variants family into an SPL and evolution of SPLs [5]. Regarding SPLs, a prerequisite for systematic reuse is to explicitly specify evolutionary change of variants family in a variability model [6]. Feature models (FMs) constitute a suitable to model address migration and evolution of SPLs [3]. Depending on the abstraction level, features may refer to a prominent or distinctive user-visible characteristic or functionality of variants family [7, 8].

Actually, SPLs have been commercially applied in many industry domains [9]. Related to the challenge of adopting SPLs taking care of their evolution in the automotive industry, initially migration of automotive variants family into an SPL often starts with an analysis of variability [5]. The automotive industry faces challenges to manage variability among variants family, due to its product domain. Numerous research papers point out the necessity of requirements and system modelling to manage variability in the automotive industry [10]. Most proposals in this context do not consider textual requirements, even though they enable to manage variability from the beginning [11][12].

This work proposes an approach to address the use of SPLs at the requirements level. The purpose of this paper is to recommend a practical evolution-based approach that supports a reengineering process to migrate automotive variants family into an SPL and to evolve an SPL after it has been established. This research work applies the proposed approach to the CSVF at Bosch Company and analyses the approach through a case study involving the CSVF and the classical sensor development team (CSDT). The approach includes different activities for both the reverse and forward

engineering chaining phases and it provides guidelines for the CSDT in an SPL context for the automotive industry. The focus of this work remains at the requirements-level.

2 Related work

Despite the research works that address adoption of SPLs [2, 13], there is still a lack of the approaches that propose guidelines or methods for performing SPLs migration and evolution in a systematic way. However, most of the presented approaches were evaluated using toy examples or open-source applications (e.g., ArgoUML) [14, 15]. Few approaches were evaluated using industrial case studies [16, 17].

In addition, other works propose refactoring patterns and notations that fit the SPLs context. Moreover, they are applicable to FMs [18, 19]. As a common practice, the authors in [18, 19] evaluate their work in mobile applications and health information domains respectively. Another work [20] proposes a generic framework for managing collections of related cloned products into an SPL, where the products are refactored into a single-copy SPL. Empirically, this work analyses three industrial case studies of different organizations.

A systematic study provides an overview of current research on reengineering of existing systems into SPLs [21]. This study concludes that reengineering of existing systems into SPLs is an active research topic with real benefits in practice. Moreover, it motivates new research in the adoption of systematic reuse in software companies. In this context, it reports the lack of sophisticated refactoring, the need for new metrics and measures, and more robust evaluation.

A model-driven approach to support software engineers in handling source code variability of software variants in the automotive domain is proposed in [22]. This work contrasts with ours, since it does not consider the requirements of the automotive domain, when modelling and managing variability [22].

Close to our work, the authors in [23] introduce a systematic reuse method called Variation Point Method (VPM), which models variability in a process that starts with common requirements. Important research works related to managing variability are [4, 8, 24-28].

However, not only is our approach designed to consider the variants that are related to the same family in the automotive domain but also it is used to address commonality and variability of variants family at a higher level of abstraction using variability model (i.e., an FM). Moreover, it is planned to use SPLs refactoring. In addition, our approach is analysed and evaluated in an industrial case study.

3 Our approach

Based on the general process of evolution of SPLs and the analysis of our industrial case study from the automotive domain, we propose EVOA-SPL, an evolution-based approach for SPLs. EVOA-SPL supports the evolution of an SPL focusing on migration of the variants into an SPL. To tackle this challenge, the approach considers variability at the requirements-level. EVOA-SPL supports a reengineering process towards systematic reuse and consistent evolution of an SPL.

As shown in Figure 1, EVOA-SPL adopts a reengineering process, which consists of two distinct phases: the reverse engineering phase and the forward engineering phase. Each of them is summarily described in the following subsections. The FM works as a common model that is shared between two phases. Phase 1 derives an FM and phase 2 upgrades and refines it. The adoption of a reengineering process is amenable for EVOA-SPL, since it considers existing variants and it is incremental (support a new change to the requirements in a systematic way).

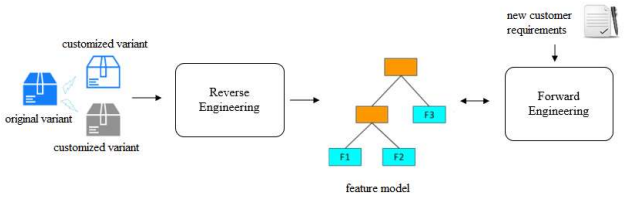


Figure 1: The EVOA-SPL approach phases.

3.1 The reverse engineering phase

The reverse engineering phase consists of three main activities: (1) the difference analysis, (2) the variability analysis, and (3) the feature model synthesis. The main input for this phase are the textual requirements of each two variants. Thus, this phase identifies commonality and variability among automotive variants family at the requirements-level. It uses the textual requirements of only two variants and derives an FM. The project manager and domain expert may be consulted to clarify any information about the variants; to select proper textual requirements from the document, and to revise and confirm the derived FM.

3.2 The forward engineering phase

The forward engineering phase consists of three main activities: (1) the bootstrapping, (2) the evolution, and (3) the FM refactoring. The reverse engineering phase delivers an FM that is used as an input for this phase. The activities of this phase use the current version of the FM and the textual requirements of another / new variant to evolve the SPL.

The bootstrapping activity imports variants family (one by one) according to the decisions of the domain experts and the project manager. Once an SPL has been bootstrapped, the evolution process can start. It evolves an SPL with a new variant upon receiving a new customer request. Both activities derive and store features of a (new) variant into a features list (FL) and refactor (i.e., refine) the current FM with the requirements (features) of a (new) variant that is not supported yet by the SPL.

3.3 EVOA-SPL activities

In order to migrate the variants into an SPL and then support an SPL evolution, EVOA-SPL goes through a reengineering process. EVOA-SPL uses the textual requirements of automotive variants, which were initially created with ad-hoc approaches. Before performing the EVOA-SPL activities, one is required to study the domain-specific issues related to automotive variants family, like notations, technique, or process steps [29]. The EVOA-SPL activities that the software engineers can follow to migrate automotive variants family into an SPL and to support its evolution are the following (see Figure 2).

3.3.1 Activity 1. The **difference analysis** activity captures and identifies similarities and differences between two variants. This activity (i) writes textual requirements of each variant into atomic requirements (ARs), (ii) gives each AR a unique id, and (iii) stores ARs in the so-called requirements document (RD). Furthermore, (iv) it uses a proper text-based comparison tool to specify common and optional ARs among several RDs. The comparison concerns changes between RDs in terms of matched, added, deleted, and modified ARs. Figure 3 depicts the RD structure at the end of this activity.

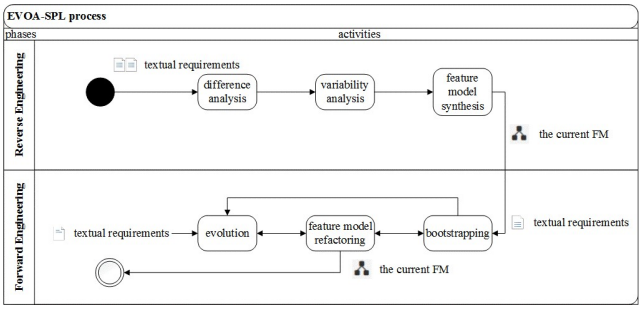


Figure 2: EVOA-SPL activities.

3.3.2 Activity 2. The **variability analysis** activity performs further processing and identifies commonality and variability among variants. The main inputs of this activity are RDs of two variants. The variability analysis activity (i) specifies common and optional ARs from RDs of two variants and (ii) stores those classified ARs in a single master document (SMD). It represents an initial adequate view of commonality and variability among automotive variants family, since variant members are related to the same family and shared the same domain architecture.

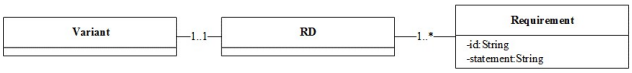


Figure 3: The structure of RD.

Additionally, this activity (iii) uses the "common", "optional", "OR-Group", and "XOR-Group" keywords to classify the variability-pattern of each AR in an SMD (please see Table 1). The variability analysis activity (iv) uses text-parsing to extract the set of keywords of each AR in an SMD that have important meaning to identify features and their dependencies. It parses actions (verbs), objects including instruments, technologies, services, parameters (attributes) and signal names. These keywords set helps to identify feature names and their dependencies. This activity (v) suggests a proper name for the feature using the keywords set and based on the suggestions from the domain experts. Next, this activity identifies features and their variability-pattern (please see Table 1) using the feature identification method (see Section 3.4). Finally, this activity (vi) transfers and organizes variability information into an FL, which contains a set of features, each one with a predefined variability-pattern and a feature-relationship. A feature-relationship can have two types; parent-child relationship and dependency relationship. Moreover, in this list, a feature is related to a set of ARs that are responsible for its specification in an SMD.

Table 1: The feature variability-pattern.

Common	The feature must be included in every variant.
Optional	The feature may be included (or not) in every variant, but it is not necessary or required
OR-Group	The feature is part of a group of features and if their parent is included, at least one of those features is included in the variant.
XOR-Group	The feature is part of a group of features, and if their parent is included exactly one of those features is included in the variant.

3.3.3 Activity 3. The **feature model synthesis** activity synthesizes an FM. This activity maps variability information from an FL to an FM and delivers a model that contains: feature name, variability-pattern, and feature-relationship. One of the relationships that needs to be captured is a parent-child relationship to show that one feature is a child of another one. The other type of relationship is dependency relationship; requires dependency means "feature A requires feature B" to be selected and excludes means "feature C requires feature D" to not be selected. In other words, both features C and D should never be simultaneously used in the same variant.

To build an FM, this activity (i) reads features from an FL sequentially. Then it (ii) uses mapping table to transform from an FL into an FM (please see Table 2). The mapping table relates variability notations between an FL and an FM. Finally, this activity (iii) builds the FM with a given modelling tool (e.g., "FeatureIDE") [12].

To build the FM, the software engineers (i) draws features inside the respective symbol (rectangle), (ii) defines variability-pattern for each feature, and (iii) draws child-feature under their parent-feature and defines feature-relationship among them.

Table 2: The mapping notations between an FL and an FM.

feature list	feature model	
title		root feature
feature		Feature
common feature		mandatory feature
optional feature		optional feature
OR-Group		Or
XOR-Group		alternatives
parent-child relationship		parent-child relationship

3.3.4 Activity 4. The **bootstrapping** activity adds remaining members of variants family (one by one) to contribute to an SPL. This activity (i) uses textual requirements of a given variant, (ii) derives features of that variant, and (iii) stores them in an FL. Finally, this activity (iv) evolves current FM to encompass features of the variant using *an FM refactoring scenario* (see feature model refactoring activity).

3.3.5 Activity 5. The **feature model refactoring** activity refactors and refines the current FM with features of a (new) variant that is not supported yet by the SPL. This activity uses a catalogue of sound FM refactorings to perform transformations that improve and increase the current FM.

These refactorings appear in [18]. Fortunately, these refactorings are reasoned and proved in [18]. The feature model refactoring activity uses *an FM refactoring scenario* that works as follow. It (i) reads a feature from an FL (top to down) and (ii) matches feature (and its variability-pattern) with nodes (and their variability-

pattern) of current FM. Based on the match-status, the scenario (iii) performs the proper an FM refactorings. *An FM refactoring scenario* considers the following match-status:

1. Status 1: a feature requires changes in current FM (please see Table 3). This leads to apply to the current FM the proper refactoring that appear in [18].
2. Status 2: a feature appears in the current FM with the same variability information. This leads to keep the current FM unchanged. Thus, the refactoring are only applied in case of differences between an FL and an FM.
3. Status 3: a feature appears as a common feature in the current FM and does not appear in an FL of a (new) variant. This leads to apply the proper refactoring that appear in [18] to the current FM, in order to transform the common feature into an optional feature.

An FM refactoring scenario repeats until reaching the final feature in an FL. It is worth to mention that if evolution is a feature deletion, *an FM refactoring scenario* of the EVOA-SPL approach does not support this case. Since it needs the requirement engineer to check and approve this evolution. In addition, it requires not only to delete feature from current FM but also to remove the associated components and feature-software-unit.

Table 3: The feature changes in the current FM.

1.	Add a new common feature.
2.	Add a new optional feature.
3.	Add a new Or (OR-Group) feature.
4.	Add a new alternative (XOR-Group) feature.
5.	Transform an optional feature into a common feature.

3.3.6 Activity 6. The **evolution** activity propagates requirements (features) of a new variant to the current FM. Actually; this activity evolves an SPL to encompass a new variant once it has been bootstrapped. This activity (i) uses textual requirements of a new variant, (ii) derives features for this variant, and (iii) stores them in an FL. Finally, this activity (iv) evolves the current FM to encompass features of a new variant using *an FM refactoring scenario* (see feature model refactoring activity).

3.4 The feature identification method

We proposed the Feature Identification Method (FIM), which consists of three main interconnected parts. FIM identifies features and their variability–pattern and feature-relationship in an FL. The main parts of FIM, which is inspired by forward chaining in expert system [30], are documented below.

Part1. FIM variability knowledge (VK) is a collection of facts (see below) that are applied to each ARs sequentially.

1. A set of ARs represents a feature.
2. A set of common ARs represents a common feature.
3. A set of variable ARs represents an optional feature.
4. Grouped ARs belongs to the same main feature.
5. A main AR forms a parent-feature.
6. Grouped ARs form child-feature.
7. AR forms a child-feature when it has one value and does not carry options or alternatives.
8. AR forms OR-Group or XOR-Group when it is within a group of options. The former is used when at least one or more ARs (options) can be included in a variant and the latter is used when just one AR (option) can be included in a variant. This can be observed using an SMD.
9. AR that required another AR to be included forms a dependent relationship.
10. AR that required another AR not to be included forms an independent relationship.

Part 2. FIM variability rules (VRs) are a collection of rules (see below) that have an if-then statement format. The if-then statement consists of two sides, on the left-hand (LHS) is if-side and on the right-hand (RHS) is then-side. We can apply the rule on AR whenever a given AR matches the LHS of VR.

1. R1: If AR is a child-feature then it has a parent-child relationship with parent feature.
2. R2: If AR forms OR-Group then it has OR-Group with parent-feature.
3. R3: If AR forms XOR-Group then it has XOR-Group with parent-feature.
4. R4: If AR forms a dependent relationship with other AR then it has requires relationship.
5. R5: If AR forms an independent relationship with other AR then it has excludes relationship.

Part 3. The FIM process works on ARs of an SMD and updates them sequentially. The FIM starts the first iteration with the VK and applies them sequentially on each AR until the last AR in the SMD is reached. In case that one of the VK is not satisfied, (does not match an AR) it will be skipped. After ARs are updated by the VK, the FIM starts the second iteration with VRs. The method searches ARs until one of them matches LHS of the VRs and then applies the RHS of this VR on that AR. The FIM stops when reaches last AR in an SMD. Now an SMD is updated with variability information that makes it rich enough to feed feature model synthesis activity.

4 The case study

In order to evaluate the EVOA-SPL approach, we conducted a case study following the guidelines, which are presented in [31]. According to [31], the case study is composed of four major process steps to be walked through: planning, design, data collection and, analysis and reporting.

4.1 Planning

Good planning is necessary for the success of the case study. Therefore, we planned several issues.

Objective. The objective of the case study is to evaluate the EVOA-SPL approach in the automotive domain using the CSVF. The case study will be conducted with the CSDT who has previous experience in the automotive domain development.

Treatment. Our case study has one treatment, which is the EVOA-SPL approach.

Objects. The object of our case study is the CSVF, which are implemented based on AUTOSAR architecture. We used four

variants that are related to the CSVF, specifically, the textual requirement of each variant.

Subjects. The subjects of the study are the CSDT and the project manager.

Methods. We planned to perform our case study in two stages. In the first stage, we took the role of a software engineer and we applied the EVOA-SPL approach on the CSVF. During the execution of the EVOA-SPL approach process, initially, we derived the SMD and the FM. After that, we started to bootstrap the CSVF in the SPL and then we evolved the SPL with the new variant using *an FM refactoring scenario*.

In the second stage, we planned to evaluate the EVOA-SPL approach and the generated artefacts using several data collection methods, where we have found the empirical study including survey, interview, and observation are applicable to the environment of our case study

4.2 The variants family used in the study

The CSVF has been developed and customized by the team for more than 3 years, to satisfy the needs of different customers in the automotive domain. Moreover, they have around 20 major releases. The CSVF has source code written in C and includes 50 packages. A number of features have been added and modified, while the variants have been evolved over time. To conduct this case study, we used the CSVF, which has a fixed architecture (AUTOSAR).

The CSVF consists of three variants, which were developed according to the needs of customers in the automotive domain, and a new variant, which was an upcoming variant scheduled for being developed in the near future upon receiving a new customer request. All the variants were cloned from the original variant (platform variant, which is often evolved from the platform developed and successfully used by the first customer) and then modified according to customer needs. To simplify the following discussion, we designate for every variant in the CSVF a number and we called the upcoming variant a new variant.

The subjects involved in the study had full access to the variants family documentation and the code. Even though we took the role of software engineers, we applied the EVOA-SPL approach on CSVF artefacts; we had limited access to the documentation and code. For confidentiality, no all details of the CSVF are provided.

4.3 Data Collection

Case Study Environment. The study was conducted in the software development department, during October 2017- June 2018 at Bosch Company.

Procedure. The study was conducted in two stages.

Stage 1. We took the role of a software engineer and we applied the EVOA-SPL approach on the CSVF using the textual requirement (i.e., requirement specification document) of each variant.

Stage 2. We evaluated the effectiveness and the efficiency of the EVOA-SPL approach from the point view of CSDT and we evaluate the correctness of the EVOA-SPL approach concerning project manager perspective.

Summary of Generated Artefacts. Following the EVOA-SPL approach, the main activities during the execution of the case study in stage 1 were difference analysis, variability analysis, feature model synthesis, bootstrapping, feature model refactoring, and evolution.

Firstly, the SMD of variants family was defined, capturing commonality and variability between variant 1 and variant 2. Secondly, the FM was derived, representing the SPL at a high level of abstraction. The model presents common features of the SPL, which are the *Message* including the *Transmit* and the *Receive* messages, the *Diagnosis*, the *Monitoring* and the optional features, which are the *Calculations* and the *Interface support*. On the other hand, the *Transmit* feature has two features involving the *Layout* and the *Signals*, where the *Layout* feature has two alternatives, which are *Layout 1* and *Layout 2*. This means that the variants have two different message layouts, one for each variant: the first layout is *Layout 1*, which has five bits; the second one is *Layout 2*, which has seven bits.

Concerning the *Signals* feature, it can be contained different signals for the variants (variant 1 and variant 2). Some of these features are common (which are *Signal 1*, *Signal 2*, and *Signal 3*), some of them are optional (which are *Signal 4* and *Signal 5*), and some of them are within alternative group (which are *Flag 1*, *Flag 2*, *Algorithm 1*, and *Algorithm 2*).

After structuring the FM, it is the time to bootstrap the CSVF into the SPL and then evolve the SPL. The former evolves the current FM with the requirements (the features) of the remaining variant of variants family (variant 3). The later evolves the current FM with the requirements (the features) of a new variant.

Firstly, the CSVF must be bootstrapped into the SPL. For that, the features of variant 3 are identified and stored in the FL. At this point, the FM is refined with the features of variant 3 using *an FM refactoring scenario*. Now the SPL is bootstrapped completely and the commonality and the variability of the CSVF are both presented by the current FM.

The *Signal 6* feature, which exists in the FL of variant 3 and does not exist in the current FM, presents new features. *Signal 6* feature appears as a new feature in the list, since it does not exist in the current FM. At the same time, this feature represents a change at the SPL level (adding a new feature). As shown in Figure 4, this change is propagated to the SPL and the new feature (*Signal 6*) is moved from the FL of variant 3 to the FM using *an FM refactoring scenario* (the scenario uses Refactoring 12 “add optional node” [18]).

Once the bootstrapping is completed, the SPL can be evolved with features of the new variant, using *an FM refactoring scenario*. Prior to this step, the features of the new variant are derived and stored in the FL of the new variant. The FL presents the features of the new variant, the *Identification* feature appears as a new type of the *Message* feature and *Flag 3* feature appears as another alternative of the *Flag* feature, since they do not exist in the current FM. These new features represent changes in the SPL level (adding new features).

As shown in Figure 4, these changes are propagated to the SPL and the new features (*Identification* and *Flag 3*) are moved from the FL of new variant to the FM using *an FM refactoring scenario*. Regarding the *Identification* feature, the scenario uses Refactoring 12 “add optional node” and regarding *Flag 3* feature, the scenario uses Refactoring 5 “add new alternative” [18].

The last step is to perform a typical documentation and readable artefacts preparation. In the end, the approach activities have been

applied to the CSVF and the FM has been derived, synthesised and evolved, as shown in Figure 4, where the refactoring locations are highlighted. At this point, stage 2 of the case study can be started to assess the EVOA-SPL approach by CSDT and the project manager.

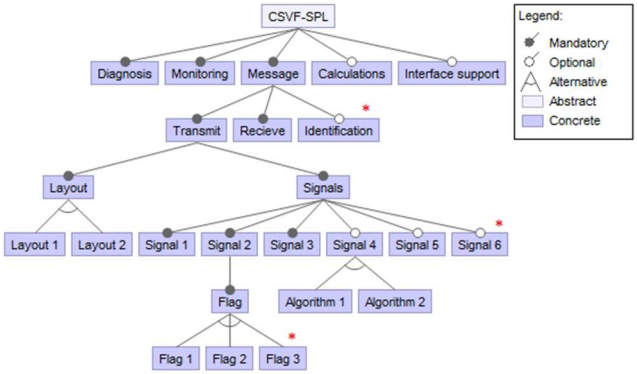


Figure 4: The current FM of the CSVF.

The objective of stage 2 of the case study is to assess the effectiveness and the efficiency of EVOA-SPL according to the CSDT and its correctness according to the project manager perspective. Hence, we formulated the following hypotheses to measure the effectiveness, efficiency, and correctness of the EVOA-SPL approach.

H1: EVOA-SPL is effective.
H2: EVOA-SPL is efficient.
H3: EVOA-SPL is correct.

4.4 Execution of Data Collection

We prepared many documents that suited to the collecting data methods. We designed a presentation that introduces the EVOA-SPL activities and the generated artefacts to the members of CSDT. In addition, we prepared a survey that is a part of an empirical study. The survey includes a questionnaire with 22 questions. These questions were formulated by using a combination of descriptive, behaviour, and attitudinal questions. The answers were given using ordinal and nominal scale response formats. One of the most important members of the team is undoubtedly the project manager. Thus, it is of special importance to dedicate her a survey, which contains measurement items related directly to the hypotheses.

For the empirical study, we defined a set of items to be evaluated by asking the CSDT members to perform specific tasks and then answer questions while using directly with the EVOA-SPL approach. Ideally, to compare the developer's solutions with a possibly-correct solution, in order to investigate our hypotheses, we have defined "correct" solutions for the tasks. The aim was to evaluate the effectiveness and the efficiency of the CSDT while applying EVOA- approach. For that, we defined five dependent variables.

Regarding effectiveness, we defined: **Effectiveness-SMD**, which is calculated as the ratio between the number of correct variability retrieval scenarios from the SMD that the CSDT identified and the total number of correct retrievals. **Effectiveness-FM** is calculated as the ratio between the number of correct feature retrieval scenarios from the current FM that the CSDT member identified and the total number of correct retrievals. **Effectiveness-EVO** is calculated as the ratio between the number of correct evolution scenarios to the current FM that the CSDT member performed and the total number of correct evolutions.

Regarding efficiency, we defined **Efficiency-SMD** as the ratio between the number of correct variability retrieval scenarios from the SMD that the CSDT member identified and the total time he spent. **Efficiency-FM** is computed as the ratio between the number of correct feature retrievals scenario from the current FM that the CSDT member identified and the total time she spent. **Efficiency-EVO** is calculated as the ratio between the number of correct evolutions to the current FM that the CSDT member performed and the total time he spent.

Regarding correctness, **Correctness-EVOA-SPL** is defined as the ratio between the number of positive feedback from the project manager about the validity of the variability information provided by EVOA-SPL and the number of survey questioners that is written and dedicated to get the feedback of project manager.

For the direct methods to collect data, we prepared an interview with the CSDT members to get direct feedback related to EVOA-SPL by establishing open questions. Moreover, in order to get a deeper understanding, we used the observation. We informed the project manager that we are going to investigate the use of EVOA-SPL by the CSDT members in their normal daily work. We agreed with them to use EVOA-SPL during two weeks (daily for one hour). To perform a data collection, the following steps were conducted:

1. We met the software developers of the CSDT (the project manager and seven developers). We started the first session by training including the presentation about EVOA-SPL and training exercises on using its capabilities.
2. We established the second session; we took four days with eight sessions, each session of one hour. We met the software developers individually (it is not recommended to simultaneously pause the work of many developers in an industrial company for a long period).
3. We performed the empirical case study. In the first 10 minutes, we took the developer background information using a form, and then we gave them 3 tasks[1] to perform using EVOA-SPL. Each task consists of 3 tags[2]. Finally, we asked the developer to answer the survey about EVOA-SPL.

Data collection through semi-structured interviews was performed using questions about a set of subjects related with EVOA-SPL. The interview dialog was guided by a set of questions. Simultaneously, we observed the CSDT members while using EVOA-SPL and we took notes about those observations..

[1] According to the policy of the company, we hide the tasks.

[2] Tag is a request for the developer to retrieve information or to make change using the EVoSPL approach.

4.5 Analysis and reporting

We performed a qualitative analysis related to the dependent variables to prove our three hypotheses H1-H3. The qualitative analysis was undertaken based on the collected data that was performed earlier. Related to the tasks that were performed by the developers within the empirical study. Table 4 summarises the results of the qualitative analysis.

Firstly, we investigate the effectiveness of task 1 (**Effectiveness-SMD**), which is related to the task of retrieving the variability information from the SMD. We compared the developer's solutions with the correct solutions. The result reveals that the developers were able to achieve a high percentage; they solved around 93% of the total tags related to this task. In what concerns the effectiveness in task 2 (**Effectiveness-FM**) of using the current FM to retrieve variability information, the CSDT members were able to retrieve correctly around 89% of the total variability information. The effectiveness in task 3 (**Effectiveness-EVO**) to perform evolution scenarios to the current FM, the CSDT members were able to perform correctly around 85% of the total evolutions.

We repeated the analysis for the same tasks, but we measured the time used and we estimated the efficiency. The results show that the CSDT members took around 10 minutes to complete task 1, which originates an **Efficiency-SMD** value of 0.26. The CSDT members took around 12 minutes to complete task 2, so *Efficiency-FM* is equal to 0.21. Finally, the CSDT members took around 15 minutes to complete task 3, with an **Efficiency-EVO** value of 0.18.

Table 4: Mean and max for the data analysis of dependent variables.

subjective dependent variable	mean	max
Effectiveness-SMD	0.93	1.00
Effectiveness-FM	0.89	
Effectiveness-EVO	0.85	
Efficiency-SMD	0.26	0.30
Efficiency-FM	0.21	0.25
Efficiency-EVO	0.18	0.20
Correctness-EVOA-SPL	0.86	1.00

We repeated the analysis for the same tasks to estimate the efficiency but this time we asked the CSDT to perform the tasks using the normal approach[3] (and its related artefacts), which is adopted by the company for a long time to develop the CSVF to satisfy customer's needs. The results show that the CSDT members took around 20 minutes to complete task 1, with an **Efficiency - SMD** value of 0.14. The CSDT members took around 30 minutes to complete task 2, with an **Efficiency-FM** value of 0.08. Finally, the CSDT members took around 28 minutes to complete task 3, with an **Efficiency-EVO** value of 0.09. Figure 5 shows a chart that compares the EVOA-SPL approach with the normal approach regarding efficiency while the CSDT performing the tasks related to the empirical study. In total, the comparison reveals that the CSDT performed the tasks related to the empirical case study more efficiently than using the normal approach.

Regarding the correctness, we depend on the feedback of the project manager. The project manager highly agreed that the results of the EVOA-SPL approach are valid (the **Correctness-EVOA-SPL** is equal to 86%). Due to the manual observation while the CSDT members were performing exercises related to EVOA-SPL, the members were able to understand and interact with the approach smoothly. They believe that the activities and the artefacts of the EVOA-SPL approach can be used to achieve the intended objectives. Moreover, the survey results reflect high satisfaction, which helped us to find a new hypothesis related to our approach (it is useful).

During the interviews, the project manager and (most of) the CSDT members confirmed that EVOA-SPL supports the move towards an SPL. Moreover, they confirmed that the approach helped to retrieve features that took many hours to search for them in the artefacts of the CSVF. Concretely, we analysed the developer's feedback that was provided using the survey questionnaire. The developers not only agreed that EVOA-SPL can be used to manage variability over the CSVF, but they also agreed that it can help to reduce the effort.

4.6 Threats to Validity

The main threat to the validity of the empirical study is the missing in the quantitative analysis. For that, we plan to perform this analysis in our future work. Moreover, the design of the survey questionnaire, the number of developers who shared in the empirical study, and the exchange of information between the developers are other main reasons that may threaten internal validity.

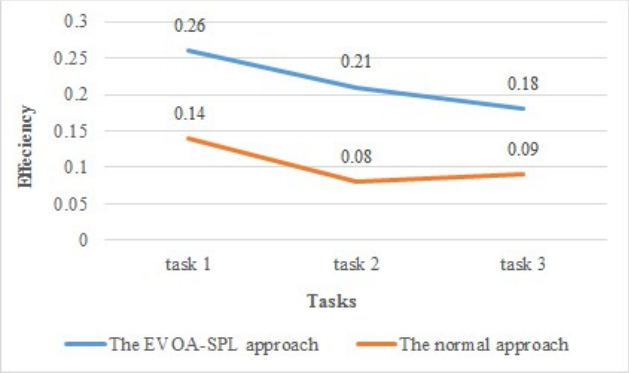


Figure 5: The comparison of the efficiency between the EVOA-SPL approach and the normal approach.

5 Conclusions

In this paper, we introduce the EVOA-SPL approach to support migration of automotive variants into an SPL and then support the evolution of the SPL, focusing on the textual requirements of the variants. We present a case study in the automotive domain conducted for the classical sensor variants family at Bosch Company, using the guidelines described in [31]. The results of the case study have shown that EVOA-SPL can be suitable for the automotive industry.

As future work, we need to perform additional empirical evaluation with larger and more complex SPLs. We want also to improve/enrich the approach by reducing the number of activities and the number of steps within each activity. Moreover, we plan to build a tool to automate the manual steps of EVOA-SPL.

[3] For confidentiality, many details of the normal approach are not provided

ACKNOWLEDGMENTS

The University of Minho and Bosch Company supported this research. We thank our colleagues from the classical sensor development team at Bosch Company. Especially André L. Ferreira and Jana Seidel for their active collaboration and support. Special acknowledgment to the spirit of Helder Boas, who passed away after he offered the help and support to this research work.

REFERENCES

[1] Vander Alves, Nan Niu, Carina Alves, and George Valença (2010). Requirements engineering for software product lines: A systematic literature review. *Information and Software Technology*, 52(8), 806-820.

[2] 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 *Proceedings of the 19th Int. Conf. on Software Product Line*, 101-110.

[3] 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 *Int. Conf. on Software Reuse*, 302-307.

[4] Fernando Wanderley, Denis Silva da Silveira, João Araujo, and Maria Lencastre (2012). Generating feature model from creative requirements using model driven design. In *Proceedings of the 16th International Software Product Line Conference-Volume 2*, 18-25.

[5] Andreas Metzger and Klaus Pohl (2014). Software product line engineering and variability management: achievements and challenges. In *Proceedings of the Future of Software Engineering*, 70-84.

[6] Klaus Pohl and Andreas Metzger (2006). Variability management in software product line engineering. In *Proceedings of the 28th Int. Conf. on Software Engineering*, 1049-1050.

[7] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake (2016). Feature-oriented software product lines. Springer-Verlag.

[8] Jean-Marc Davril, Edouard Delfosse, Negar Hariri, Mathieu Acher, Jane Cleland-Huang, and Patrick Heymans (2013). Feature model extraction from large collections of informal product descriptions. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering,* 290-300

[9] Mikael Svahnberg and Jan Bosch (1999). Evolution in software product lines: Two cases. *Journal of Software Maintenance: Research and Practice,* 11(6), 391-422.

[10] Olivier Renault (2014). Reuse/Variability Management and System Engineering. In *Poster Workshop of the Complex Systems Design & Management Conference CSD&M 2014*, 173.

[11] Yang Li, Sandro Schulze, and Gunter Saake (2017). Reverse engineering variability from natural language documents: A systematic literature review. In P*roceedings of the 21st International Systems and Software Product Line Conference-Volume A*, 133-142.

[12] Daniela Rabiser, Paul Grünbacher, Herbert Prähofer, and Florian Angerer (2016). A prototype-based approach for managing clones in clone-and-own product lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*, 35-44.

[13] Samuel A. Ajila and Patrick J. Tierney (2002). The FOOM method-modeling software product lines in industrial settings. In *Proceedings of the 2002 Int. Conf. on Software Engineering Research and Practice*.

[14] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. (2011). Extracting software product lines: A case study using conditional compilation. In *15th European Conference on Software Maintenance and Reengineering*, 191-200.

[15] Jabier Martinez, Tewfik Ziadi, Tegawende F. Bissyande, Jacques Klein, and Yves Le Traon (2015). Automating the extraction of model-based software product lines from model variants. In *30th IEEE/ACM Int. Conf. on Automated Software Engineering*, 396-406.

[16] Vander Alves, Pedro Matos, Leonardo Cole, Paulo Borba, and Geber Ramalho (2005). Extracting and evolving mobile games product lines. In *Int. Conf. on Software Product Lines*, 70-81.

[17] Bo Zhang and Martin Becker (2012). Code-based variability model extraction for software product line improvement. In *Proceedings of the 16th International Software Product Line Conference-Volume 2*, 91-98.

[18] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena (2006). Refactoring product lines. In *Proceedings of the 5th Int. Conf. on Generative Programming and Component Engineering*, 201-210.

[19] Mohammad Tanhaei, Jafar Habibi, and Seyed-Hassan Mirian-Hosseinabadi (2016). A feature model based framework for refactoring software product line architecture. *Journal of Computer Science and Technology* 31(5), 951-986.

[20] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik (2013). Managing cloned variants: a framework and experience. In *Proceedings of the 17th International Software Product Line Conference*, 101-110.

[21] Wesley KG 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), 2972-3016.

[22] Cem Mengi, Christian Fuß, Ruben Zimmermann, and Ismet Aktas (2009). Model-driven Support for Source Code Variability in Automotive Software Engineering. In *1st MAPLE Workshop*, 44-50.

[23] Diana L. Webber and Hassan Gomaa (2004). Modeling variability in software product lines with the variation point model. *Science of Computer Programming* 53(3), 305-331.

[24] Hitesh Yadav and A. Charan Kumari (2018). Analysis of Features using Feature Model in Software Product Line: A Case Study. *International Journal of Education and Management Engineering* 8(2), 48-57.

[25] 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 *15th IEEE Int. Requirements Engineering Conference (RE 2007)*, 243-253.

[26] Jabier Martinez, Tewfik Ziadi, Jacques Klein, and Yves Le Traon (2014). Identifying and visualising commonality and variability in model variants. In *European Conference on Modelling Foundations and Applications*, 117-131.

[27] Nili Itzik, Iris Reinhartz-Berger, and Yair Wand (2015). Variability analysis of requirements: Considering behavioral differences and reflecting stakeholders' perspectives. *IEEE Transactions on Software Engineering*, 42(7), 687-706.

[28] Steven She, Uwe Ryssel, Nele Andersen, Andrzej Wąsowski, and Krzysztof Czarnecki (2014). Efficient synthesis of feature models. *Information and Software Technology* 56(9), 1122-1143.

[29] Matthias Weber and Joachim Weisbrod. 2002. Requirements engineering in automotive development-experiences and challenges. In *Proceedings IEEE Joint International Conference on Requirements Engineering*. IEEE, 331-340.

[30] Ashwini Rupnawar, Ashwini Jagdale, and Samiksha Navsupe (2016). Study on Forward Chaining and Reverse Chaining in Expert System. *Int. Journal of Advanced Engineering Research and Science*, 3(12), 60-62.

[31] Johan Linaker, Sardar Muhammad Sulaman, Martin Höst, and Rafael Maiani de Mello (2015). Guidelines for Conducting Surveys in Software Engineering v. 1.1.

Analyzing Variability in Automation Software with the Variability Analysis Toolkit

Alexander Schlie
a.schlie@tu-braunschweig.de
Technische Universität Braunschweig
Braunschweig, Germany

Kamil Rosiak
k.rosiak@tu-braunschweig.de
Technische Universität Braunschweig
Braunschweig, Germany

Oliver Urbaniak
o.urbaniak@tu-braunschweig.de
Technische Universität Braunschweig
Braunschweig, Germany

Ina Schaefer
i.schaefer@tu-braunschweig.de
Technische Universität Braunschweig
Braunschweig, Germany

Birgit Vogel-Heuser
vogel-heuser@tum.de
Technische Universität München
München, Germany

ABSTRACT

Control software for *automated production systems (aPs)* becomes increasingly complex as it evolves due to changing requirements. To address varying customer demands or altered regulatory guidelines, it is common practice to create a new system variant by copying and subsequently modifying existing control software. Referred to as *clone-and-own*, proper documentation is typically not cherished, thereby entailing severe maintenance issues in the long-run. To mitigate such problems and to reinstate sustainable development, respective software systems need to be compared and their variability information needs to be reverse-engineered. However, recent work identified variability management in the domain of aPs to remain a challenging endevour and appropriate tool support to be missing.

We bridge this gap and introduce the *Variability Analysis Toolkit (VAT)*, an extensible platform that allows for the customizable definition of metrics to compare IEC61131-3 control software variants as well as providing means to visualize results. The VAT facilitates a working environment that allows for the exchange of produced results between users. By that, we aim to support engineers in re-engineering control software systems by providing them with means to define metrics based on their individual demands. We demonstrate the feasibility of the VAT using 24 software system variants implemented in accordance to the IEC61131-3 standard.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software reverse engineering**; **Software evolution**; *Software maintenance tools.*

KEYWORDS

Software Product Lines, Variability, Legacy Systems, Automation Software, Tooling

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342408

ACM Reference Format:
Alexander Schlie, Kamil Rosiak, Oliver Urbaniak, Ina Schaefer, and Birgit Vogel-Heuser. 2019. Analyzing Variability in Automation Software with the Variability Analysis Toolkit. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3307630.3342408

1 INTRODUCTION

In the domain of automated production systems, plant machinery and their control software may remain operational for decades [26]. Such systems are usually controlled by *programmable logic controllers (PLCs)*, which are typically programmed in compliance with the IEC61131-3 standard [11]. Within such lifespan, it is improbable for engineers to foresee the entire scope of functionality [18]. Changing customer requirements or altered regulatory guidelines require control software to be adapted frequently [4, 13]. To cope with changing requirements, it is common practice to copy and subsequently modify existing system implementation, an approach typically referred to as *clone-and-own* [16]. This straight-forward reuse approach saves time and workload in the short-run [18].

However, with proper documentation typically not cherished during *clone-and-own*, severe problems are induced for the long-run [22]. With a proliferation of redundant and almost-alike system variants, maintainability and evolution of the portfolio are adversely affected [3, 17, 18]. Consequently, *clone-and-own* can be a driving factor for technical debt [6, 10]. To rectify the situation and to reinstate sustainable development, respective software systems need to be analyzed and their variability information needs to be reverse-engineered [8]. For PLC control software and precisely, the family of five programming languages defined in the IEC61131-3 standard, the complexity of this task is only amplified.

To this end, recent work identified variability management in the domain of aPs to remain a challenging endevour and appropriate tool support to be scarce [7, 26, 27]. We bridge this gap and propose the Variability Analysis Toolkit *(VAT)*, a first version of a comprehensive framework to facilitate a fine-grained comparison of IEC61131-3 control software, providing customizable metrics to capture common and varying system parts. With the VAT, we provide means to display identified results in the form of *150% models*, which are considered an intuitive concept to represent variability [12]. By that, we aim to support users in facilitating strategic reuse. We make the following contributions:

- We introduce the VAT, an extensible framework, which allows for IEC61131-3 control software projects to be compared and identified variability information to be displayed.
- We demonstrate the feasibility of the VAT using 24 variants of IEC61131-3 control software provided with the *Pick-and-Place Unit (PPU)* [23], a universal demonstrator for studying evolution in aPs and report on our experiences with the VAT and PPU system variants.

The paper is structured as follows. We provide background on PLC control software and the PPU, a demonstrator for studying evolution of aPs in Sec. 2. We introduce the VAT and detail its capabilities and overall workflow in Sec. 3. We demonstrate the feasibility of the VAT and assess variants of the PPU in Sec. 4. We state related work in Sec. 5. We conclude our paper and outline future work in Sec. 6.

2 PRELIMINARIES

In this section, we provide background on PLC control software implemented in compliance with the IEC61131-3 standard and the PPU, a demonstrator for studying evolution in aPs.

2.1 IEC61131-3 Control Software

In the domain of aPs, the IEC61131-3 standard and the five programming languages it comprises are predominantly used to program control software systems for PLCs. Specifically, the IEC61131-3 language family contains three graphical programming languages, *Function Block Diagram (FBD)*, *Sequential Function Chart (SFC)* and *Ladder Diagram (LD)*, as well as two textual programming languages, *Instruction List (IL)* and *Structured Text (ST)*. For PLC software projects, the IEC61131-3 additionally determines their overall structure by defining the projects building blocks and their relations. Specifically, an IEC61131-3 software project consists of *tasks*, which control the operation of *Program Organization Units (POUs)*, which form the projects main building blocks. POUs are either a *Function Block (FB)*, a *Program (PRG)* or a *Function (FUN)*. Data exchange between POUs is realized via *actions* or *global variables*. *Actions* are a specific functionality provided by a POU for other POUs to *call*, i.e. to read another POU's internal *variables*. Each POU is implemented in one of the five programming languages of the IEC61131-3 standard. Thereby, multiple programming languages may be used in combination within one PLC project, with each POU being dedicated to realize certain software system functionality. Furthermore, the five programming languages of the IEC61131-3 standard may be used in an combined fashion. For instance, a POU may be implemented in a graphical programming language, such as SFC, while the POUs actions are implemented in a textual language, such as ST. Hence, IEC61131-3 control software projects are highly diverse with respect to the mix of programming languages. Different developer environments exits to implement respective systems and exchange formats such as *PLCopen*[1] have been standardized, transforming systems to *extensible markup language (XML)* form. Hence, such systems can be distributed independently of specific vendors.

[1]PLCopen® - https://www.plcopen.org/ - May 2019
[2]Supplementaries - https://www.isf.cs.tu-bs.de/cms/team/schlie/material/VAToolkit
[3]Eclipse Foundation™ - https://wiki.eclipse.org - May 2019
[4]Eclipse Foundation™ - https://wiki.eclipse.org/Rich_Client_Platform - May 2019
[5]Oracle® - https://oracle.com/java - May 2019

2.2 The Pick-and-Place-Unit

The Pick-and-Place Unit [23], which we use as a case study throughout this paper, represents a universal demonstrator for studying evolution in aPs. The evolution history of the PPU comprises more than twenty different software projects, henceforth referred to as *scenarios*. Implemented in accordance with the IEC61131-3 standard, respective scenarios are implemented utilizing the programming languages *Structured Text* and *Sequential Function Chart*. The PPU itself, initially developed on the ISA88 standard [24], can be seen as a set of equipment modules, which can further contain control modules. Further documentation on the PPU and its evolution can be found in [23]. We depict two scenarios in Fig. 1 [23] with the initial scenario *Simple* shown left and the evolved scenario *Advanced* shown right. The latter introduces new system capability, here the *equipment* module *Stamp*, which induces the underlying software to adapt in response to the hardware change. Specifically, with the evolved scenario, the *Crane* delivers workpieces to the *Stamp*, which are then processed prior to the *Crane* transporting them to the *Slide*.

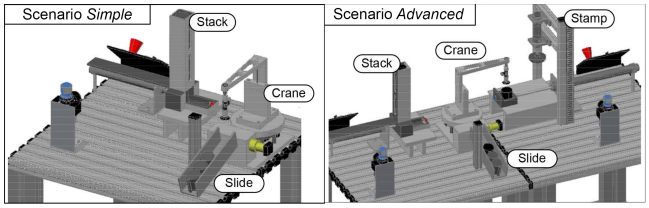


Figure 1: Two Evolution Scenarios of the PPU

3 THE VARIABILITY ANALYSIS TOOLKIT

In this section we introduce the VAT, a framework to transform PLCopen projects into model-based form, to define and maintain comparison metrics, to compare PLCopen projects and to visualize and exchange results between users. Our VAT can be found online[2], where we further provide supplementary material.

3.1 The Toolkit in a Nutshell

The VAT is based on Eclipse[3] and its *rich client platform (RCP)*[4], which results in Java[5] being the only requirement to run the VAT. We provide an overview on the VATs user interface in Sec. 3.2 and show the overall workflow to compare PLCopen projects in Fig. 2. First, *PLCopen* projects in XML form are imported and transformed into model-based form (cf. Sec. 3.3) using parsers implemented in the VAT. Such model-based representation allows for the comparison of individual projects utilizing metrics and to visualize produced results in various ways. The VAT provides a library comprising multiple comparison attributes for different project entities, such as POUs and individual IEC61131-3 programming languages (cf. Sec. 3.4). Among others, the *name* or *type* of POUs can be compared, whereas for the comparison of *variables* specifically, their *location* can be compared also. Thereby, we aim to provide users with appropriate means to customize their metrics with respect to granularity and level of detail. Utilizing a previously created metric, two projects can then be subject to the variability analysis, in which project artifacts are *compared*, *matched* and *merged* (cf. Sec. 3.5). The result is a *150% model* [20], precisely a *family model* [28], which

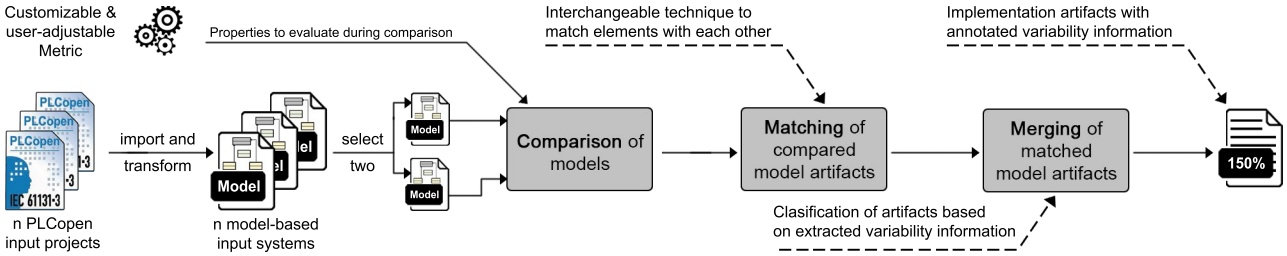


Figure 2: Schematic Workflow of the Comparison of IEC61131-3 projects with the VAT

comprises analyzed project artifacts and their annotated variability information. Generally, artifacts produced by our VAT are designed to be persistent, exchangeable and comprehensible. Hence, metrics, projects and created results are stored persistently and can be exchanged between users. To support comprehensibility, produced results preserve detailed information on how they were derived.

3.2 VATs Graphical User Interface

We depict the graphical user interface of the VAT in Fig. 3 and highlight its four main components, which are (1) the *navigation bar*, (2) the *project explorer & compare engine*, (3) the *metric manager & visualization* and (4) the *attribute manager & weight control*. Generally, the position and size of individual components are not fixed but users can resize, move, detach and attach them freely.

The *navigation bar* (cf. 1 in Fig. 3) contains menus, which allow for *PLCopen* projects to be imported and/or parsed directly. Moreover, users can set *preferences* to further customize the VAT. For instance, users can define which artifacts to store and by adjusting various thresholds, modify internal techniques such as the comparison of projects according to their individual requirements.

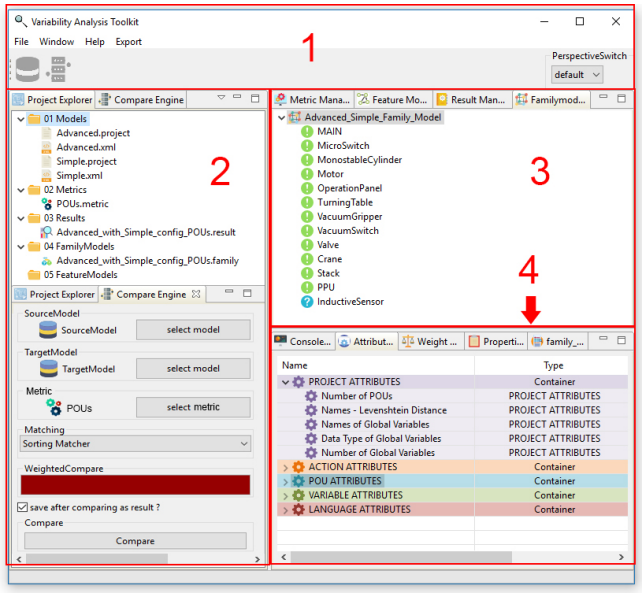


Figure 3: Graphical User Interface of the VAT

The *project explorer* (cf. 2 in Fig. 3) reflects the working environment in which artifacts are stored. The *project explorer* contains individual projects, as well as defined metrics and produced results, such as the family model. Furthermore, the project explorer facilitates a straight-forward selection of projects to trigger their comparison.

The *compare engine* (cf. 2 in Fig. 3) facilitates the comparison process and can also be adjusted further. For instance, users can introduce different *matching* techniques (cf. Fig. 2), thereby modifying the behavior of the overall comparison process if needed.

The *visualization area* (cf. 3 in Fig. 3) allows for the representation of produced results. For instance, Fig. 3 shows a family-model, which, along with other results, can be reloaded on demand. Moreover, the *metric manager* facilitates means to create metrics, which can be fully customized using the *attribute manager*.

The *attribute manager* (cf. 4 in Fig. 3) constitutes a library of atomic comparison attributes, which can be added and removed from any metric dynamically. The *weight control* allows users to state the relevance of each attribute for the comparison separately.

3.3 A Meta-Model for IEC61131-3 Projects

To facilitate a precise comparison of individual PLCopen projects, a fine-grained model-based representation is required. The VAT comprises a set of meta-models to capture the scope of PLCopen projects, which represent IEC61131-3 software systems. We depict a schematic overview of our meta-model architecture in Fig. 4 and refer to our supplementary material[2] for further details.

We provide one meta-model to reflect information regarding a projects overall setup, such as included *tasks*, contained *global variables* and comprised POUs (cf. Sec. 2). We show the meta-class for POUs in Fig. 4 and by dashed lines, indicate its' containment in the meta-model for the project setup.

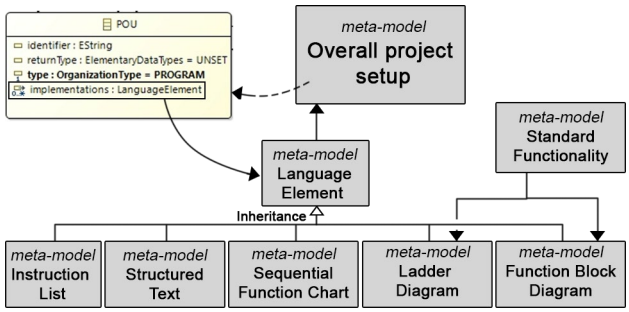


Figure 4: Schematic Overview of the Meta-Model

Furthermore, POUs are implemented in a certain programming language. Fig. 4 shows the POU meta-class to have an implementation, thereby a *Language Element*. Referenced by a solid line, the respective meta-model is *abstract* and upon instantiation, is replaced by a either programming language, such as ST or SFC. We provide a separate meta-model for each of the five programming languages of the IEC61131-3 standard. Hence, for a specific POU, its associated implementation is also captured. For instance, our meta-model for *Structured Text* provides means to capture individual statements and, furthermore, individual operations. By that, we aim to capture specific implementation details to allow for their detailed comparison in the VAT. Moreover, Fig. 4 shows *Standard Functionality* as defined in the IEC61131-3 standard, i.e., generic mathematical operations, to be captured in a separate meta-model. Solid lines in Fig. 4 illustrate *references*, i.e., *Standard Functionality* being referenced in *Ladder Diagram* implementations.

In general, our meta-models and, precisely their *classes* allow to capture detailed system artifact information including all five programming langauges of the IEC61131-3 standard. By that, we aim to facilitate a precise comparison of projects and a fine-grained analysis of their variability information. Overall, our VAT contains 8 meta-models models, totaling 52 meta-classes.

3.4 A Customizable Comparison Metric

The VAT allows for IEC61131-3 projects to be compared using a fine-grained and fully customizable metric, for which we list an excerpt from in Tab. 1. We refer to our online[2] material for more information. Within the VAT, metrics can be customized by users with respect to the level of granularity and level of detail they prefer to evaluate. For instance, a high-level overview of the projects primary building blocks and, thus, POUs, may be preferred at first. Our metric provides a respective *category* (cf. M_P in Tab. 1), in which *comparison attributes* can then be defined specifically for POUs. Such attributes can be seen as atomic comparison units, which evaluate a single property, i.e., the name of a POU and compare those in isolation. Within the metric, categories reflect *sub-metrics*, which contain properties specific to the corresponding entity, such as POUs, *actions* or *variables*. In principle, every category can be evaluated in isolation. Every category can comprise various attributes, for which some are specific to the respective category, while others are used across category boundaries. For any category, respective attributes can be included or removed dynamically, based on the users' demands. Moreover, we also allow categories to be nested, for instance, to compare POUs while taking into account their content, such as the *actions* they comprise. Therefore, certain attributes within categories act as references to *sub-metrics*, which can also be included or removed dynamically based on users' demands.

For instance, Tab. 1 shows that for *projects*, their contained POUs can be evaluated, while the category for POUs further enables an assessment of their *actions*. By nesting metrics, we aim to facilitate a comprehensive comparison of projects to account for their various entities on different levels. Every attribute *attr* calculates a similarity value ω between 0.0 and 1.0 to indicate the similarity of the compared entities regarding the analyzed property. For every attribute, its *weight*, thereby its significance for the overall comparison result, can be adjusted individually using the *weight control*. For every *category*, the sum of all weights must be *100%*.

We depict the *metric manager* and *weight control* in Fig. 5 and show (1) how individual attributes, which have been added can be selected for comparison and (2) the assignment of their respective weights. Fig. 5 shows that POUs are compared on the basis of their *type* and *name* (cf. Sec. 2). Moreover, their *actions* are considered and the *weight controller* shows their relevance to be at 20%.

With the *attribute manager*, the VAT facilitates a library of available attributes, which can be used for metric definition. With the attribute manager, attributes can be added or removed dynamically, thereby refining the metric with the *weight* controller reacting automatically to the change. Moreover, our VAT allows for various metrics to be defined, which are stored in the *project explorer*, where they can be accessed, viewed and modified. By that, metrics can be exchanged between users to support strategic communication between users on produced results.

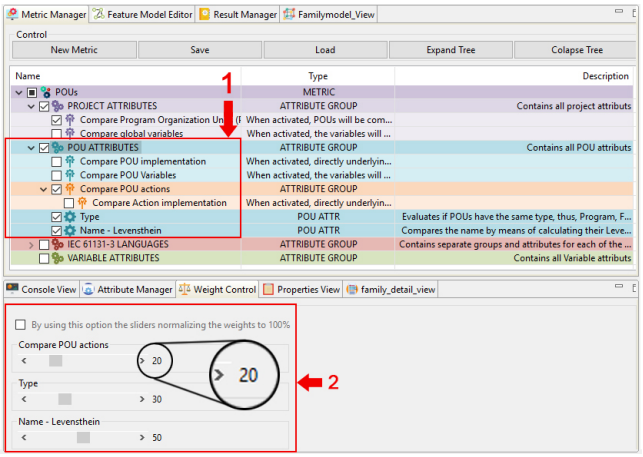


Figure 5: Metric Manager and Weight Control in the VAT

3.5 Analyzing IEC61131-3 System Variants

Within the VAT, users can select two models and a metric from the *project explorer* to start the similarity analysis. Selected models are then analyzed to the extent specified in the metric. The analysis procedure embedded within the VAT comprises three sequentially processed phases, *compare*, *match* and *merge* (cf. Fig. 2).

Compare produces pairs between artifacts of the same *categorie* (cf. Tab. 1), i.e., pairs of POUs or pairs of *actions* but no pairs between the two. For each category, we create all possible pairwise combinations. For each pair, all attributes comprised in the metric for the respective category are evaluated to compare the paired artifacts. A detailed algorithmic description can be found online[2]. For instance, comparing the POUs for the systems from Fig. 1 results in $3 \times 4 = 12$ pairs. Further considering their *actions*, all pairwise combinations of *actions* are created upon comparing two POUs. Based on weights defined by the user, a similarity value λ is calculated for each pair and, precisely, for each evaluated attribute. Similarity values are propagated upwards in a recursive fashion. For instance, the similarity of compared *actions* influences the overall similarity of the POUs they are comprised in. Hence, the similarity of two *projects* reflects the similarity of their content. Subsequently, all pairs are *filtered* to retrieve those with a high similarity value.

Table 1: Metric Categories and Exemplary Attributes to Compare IEC61131-3 Projects

Category & Attributes	References:	Description
Projects		
POUs	M_P	Compares contained POUs and references the respective sub-metric M_P
Global Variables	M_V	Compares global variables and references the respective sub-metric M_V
Name		Compares the names of projects using the Levenshtein Distance [14]
Variables (M_V)		
Scope		Evaluates if two variables are of equal scope, thus *Input*, *Output* or *Local* variables
Name		Compares the names of variables using the Levenshtein Distance
Address		Compares the location of variables.
POUs (M_P)		
Implementation	M_I	Compares the POUs implementation, thereby referencing the respective metric M_I
Variables	M_V	Compares the POUs variables, thereby referencing the metric M_V
Actions	M_A	Compares the POUs actions, thereby referencing the metric M_A
Type		Evaluates if two POUs are of the same type, hence a *program*, *function* or *function block*.
Actions (M_A)		
Implementation	M_I	Compares the actions implementation, thereby referencing the metric M_I
Name		Compares the names of actions using the Levenshtein Distance
IEC 61131-3 languages (M_I)		
Structured Text ST		Sub-metric for ST, containing various attributes, i.e. for individual *statements*
Seq. Fun. Chart (SFC)		Sub-metric for SFC, containing various attributes, i.e. for individual *steps* or *transitions*
Ladder Diagram LD		Sub-metric for LD, containing various attributes, i.e. for individual *coils* or *contacts*

During *matching*, all created pairs are filtered, thereby retrieving those with high similarity values, while removing unnecessary ones. During matching, we process every *category* separately and, precisely, in a bottom-up fashion. For instance, *actions*, which are comprised in POUs are *matched* first. Based on the removal of unnecessary comparisons, the similarity value for each POU combination is then recalculated. Subsequently, POUs are *matched*. By that, the content of an artifact is matched prior to the artifact itself, thereby creating a more sound *matching*. Precisely, all pairs within a *category* are sorted in a descending order based on their similarity value. Subsequently, the top artifact is retrieved, thereby having the highest similarity value and contained artifacts are *matched*. All remaining pairs, which contain either of the matched artifacts, are removed. Thereof, we ensure that no artifact is matched twice. Artifacts, which remain unmatched, i.e., the *Stamp* in Fig. 1 are preserved and assigned zero similarity to indicate their presence in only one of the analyzed systems.

Merge processes matched artifacts and, based on their similarity, assigns their variability, which we refer to as their *relation*. Artifacts, for which their similarity value exceeds a certain threshold λ, are classified as *mandatory*. Artifacts only present in one system are classified as *optional* (cf. *InductiveSensor* in the family model in Fig. 3). Remaining artifacts are classified as *alternative*. For implementations in either IEC61131-3 programming language, a certain change may be of minor dimension for one language but of greater one for another language. Hence, the extent of a change may differ between implementation languages and a single threshold as stated in [9] may not be universally appropriate. In general, a similarity threshold for one IEC61131-3 programming language must not be applicable to others. Consequently, the VAT lets users adjust thresholds λ for every *category* and IEC61131-3 programming language separately. By that, we aim to support customization even further.

3.6 Visualizing Results

The VAT displays comparison results in multiple forms to provide users with detailed information about compared projects. Shown in Fig. 6, the VAT yields a *150% model* (cf. 1 in Fig. 6), which depicts compared artifacts and their assigned *relation*, i.e., an *optional* output variable. For any artifact, the *detail view* (cf. 2 in Fig. 6) provides comprehensive information on all evaluated *attributes*, their specific *weights* and calculated similarities. For instance, the *container similarity* (cf. 2 in Fig. 6), which states the overall similarity of the *Cranes* implementation (cf. 1 in Fig. 6), can be related to the specific *attributes* that constitute such overall value. For ST, users can further examine identified variability by directly viewing implementation artifacts (cf. 3 in Fig. 6). Specifically, the *Cranes* implementation varies between the projects *Simple* and *Advanced* with the latter introducing an additional statement. The family model reflects that information accordingly, and marks the respective statement *optional* (cf. 1 in Fig. 6). With the *Cranes* implementation being ≈66% similar (cf. 2 in Fig. 6), the *family model* shows it to be *alternative* between the projects (cf. 1 in Fig. 6). Furthermore, we show in Fig. 6 (cf. 4) that results and can be reviewed at any time. The *result file* encapsulates the analyzed *projects*, the *metric* used for comparison, defined *thresholds* and customized *weights*. Thereby, results can be exchanged freely and examined without having either the projects or metrics. Moreover, such *result files* can be reloaded and

adjusted within the VAT, while visualizations, such as the family model, adjust automatically. The *family model* is also stored as a separate file and can be exchanged independently between users.

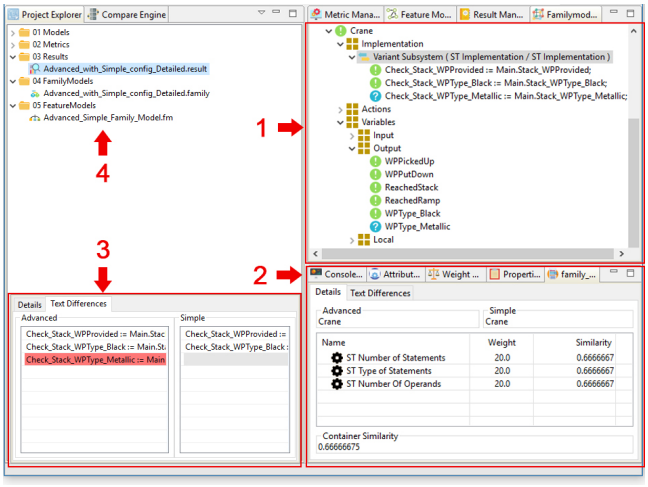


Figure 6: Detail View on Comparison Results in the VAT

4 CASE STUDY

To asses the feasibility of the VAT for comparing PLCopen projects, we use two sets of the PPU (cf. Sec. 2). Specifically, we use 13 scenarios of the PPU and 9 scenarios of the *xPPU*, with the latter being an extension of the PPU that adds further functionality. In this section, we give information on the PPU and xPPU scenarios, the used metric, state our research questions and discuss produced results.

4.1 Research Questions

With our VAT, we aim to support users in comparing *PLCopen* projects to identify common and varying parts. Thus, overall performance of our VAT should be reasonable and respective parts should be identified. Hence, we have the following research questions.

RQ1: Can we consider performance of our VAT reasonable? Especially in industry, an acceptable performance is pivotal to render a tool applicable. We refer to performance as the total runtime required for the comparison of two scenarios. Moreover, we assess its distribution between the two phases that determine the complexity of our analysis, hence, *compare* and *match* (cf. Fig. 2). Thus, we do not consider runtime to create visualizations and result files.

RQ2: Which similarity values do we yield with our VAT for the scenario sets? The overall similarity can be an indicator of the relation between projects. Hence, we evaluate the overall similarities calculated for each pairwise project comparison, separately for the two sets. We state the overall value, but specifics are online[2].

4.2 Setup

We evaluated the VAT on an Intel Core i7 (2,7 GHz) with 16GB of RAM, running Windows[6] 10 on 64bit. We list in Tab. 2 the analyzed PPU and xPPU set and information on their comprised artifacts (cf. Sec. 2). We assessed POUs, their *variables*, *actions* and ST and SFC implementations for both *actions* and POUs. The metric used for comparison assessed all such artifacts and for each *category* (cf. Tab. 1) exhibits an equal distribution of weights. We provide the full metric and all created results online[2]. Therefore, detailed similarity values can be evaluated for individual artifacts and identified variability can be assessed and examined in detail. Tab. 2 shows the 13 PPU scenarios to be in the upper portion and the 9 *xPPU* scenarios to be in the lower portion. For the PPU set, *S13* is the largest scenario, comprising 1050 artifacts captured by the VAT. For the set of 13 PPU scenarios, we performed a total of 78 pairwise comparisons, while 36 comparisons for the 9 xPPU scenarios.

[6]Windows® - https://www.microsoft.com/windows - May 2019

Table 2: PPU and xPPU Evolution Scenarios

Name	#POUs	#Variables In. | Out | Loc.	#Acts.	#Stmts. ST	#Steps SFC	#Trans. SFC
S1	13	22 | 17 | 41	50	125	95	125
S2	13	23 | 19 | 47	50	131	96	127
S3	18	39 | 24 | 74	68	194	149	196
S4a	19	39 | 24 | 75	68	194	149	196
S4b	19	39 | 24 | 78	68	194	149	196
S5	18	38 | 24 | 74	68	194	151	203
S7	19	40 | 27 | 77	69	205	152	205
S8	19	42 | 27 | 77	69	206	156	211
S9	20	49 | 33 | 86	76	261	177	236
S10	20	55 | 33 | 98	82	261	200	269
S11	20	55 | 33 | 100	81	259	202	271
S12	20	55 | 33 | 100	81	265	202	283
S13	21	55 | 36 | 103	81	268	202	283
		↑ PPU set			xPPU set ↓	
S14	21	55 | 51 | 97	85	287	212	302
S15	27	98 | 66 | 181	128	442	357	518
S16	30	113 | 74 | 191	142	481	429	614
S17	32	115 | 80 | 208	146	493	429	614
S18	33	117 | 82 | 208	146	497	429	614
S19	33	123 | 96 | 210	140	520	432	622
S20	42	127 | 94 | 210	140	525	432	622
S21	45	133 | 99 | 272	143	525	433	622
S24	46	143 | 114 | 306	157	550	465	667

In. - Input, Loc. - Local, Acts. - Actions, Stmts. - Statements in ST, Trans. - Transitions in SFC

Both sets contained the IEC61131-3 programming languages ST and SFC. In addition to all *references* from Tab. 1, we list the *attributes* used to compare projects in Tab. 3. Hence, the metric we used to compare the scenario sets from Tab. 2 comprises 16 *attributes* total.

4.3 Results and Interpretation

We can only provide aggregated data in this section and we refer to our supplementary material[2] for more information.

Performance (RQ1) For each of the scenario sets detailed in Tab. 2, we performed all pairwise comparisons and show the runtime results in Fig. 7. We show on the x-axis the total number of artifact comparisons for two scenarios and on the y-axis the required runtime in seconds $\times 10^{-2}$. Grouped in the left corner in Fig. 7 are the comparisons for the PPU set. Totaling ≈215.000 artifact comparisons at maximum, VATs runtime for comparison does not exceed one second. To compare the largest xPPU scenarios, *S21* and *S24*, with over one million compared artifact pairs, VAT requires ≈4.5 seconds. With ≈94%, the majority of runtime is required to *compare* projects, while ≈6% account for the *matching* of project artifacts.

Table 3: Metric used for Comparing Evolution Sets

Attribute Name	Description
POUs M_P	
Name	Compares names using Levenshtein distance
Type	Compares POU type (cf. Tab. 1)
Actions M_A	
Name	Compares names using Levenshtein distance
ST M_I	
# of statements	Compares the number of statements in ST
Type of statement	Compare the type of ST statements, i.e., *assignment* or *conditional*
# of operands	Compare the number of operands in ST
SFC M_I	
# of transitions	Compares the number of transitions in SFC
# of steps	Compares the number of steps in SFC
Variables M_V	
Name	Compares the number using Levenshtein distance.
Scope	Scope of variable (cf. Tab. 1)

- Number

The trend lines in Fig. 7 indicate a linear increase of runtime in relation to the number of artifact comparisons. Moreover, the total runtime required to compare the largest scenarios *S21* and *S24* and to create visualization artifacts did not exceed 10 seconds. Thus, we consider VATs performance reasonable for the evaluated sets.

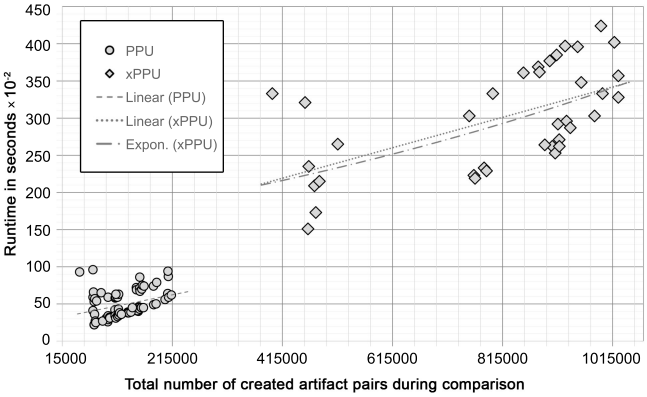


Figure 7: Runtime Results for PPU and xPPU Scenarios

Similarity Values (RQ2) For all pairwise comparisons between scenarios from the PPU and xPPU set (cf. Tab. 2), we show their calculated similarity values in Tab. 4. In each row, the similarity value decreases for each column, with only *S5* breaking the pattern for *S4a* and *S4b*. Tab. 2 shows *S5* to exhibit 18 POUs, while *S4a* and *S4b* exhibit 19 POU. Consequently, *S3* is more different to *S4a* and *S4b* than to *S5* and, thus, less similar. In general, we find the decrease of similarity values for scenario comparisons to correspond with their evolution, i.e., the scenario *S1* being more similar to *S5* than to *S8*. This patter can further be observed for the xPPU scenarios (cf. Tab. 2). Hence, the scenario *S16* is, for instance, more similar to *S18* than to *S24*. Further details on identified similarities are provided online[2]. We argue that the similarities calculated by the VAT are understandable with respect to the scenarios evolution history, during which

Table 4: Similarities in Percent for PPU & xPPU Scenarios

%	S1	S2	S3	S4a	S4b	S5	S7	S8	S9	S10	S11	S12	S13
S1	\	95.2	81.7	79.9	79.8	81.7	79.8	79.8	78.1	78.1	78.1	78.1	76.2
S2		\	84.9	83.0	82.9	84.9	82.9	82.9	80.9	80.9	80.9	80.9	78.8
S3			\	97.3	97.1	99.9	97.0	96.9	94.1	94.1	94.1	94.1	91.2
S4a				\	99.8	97.3	98.9	98.8	95.9	95.8	95.8	95.8	92.9
S4b					\	97.1	98.7	98.7	95.8	95.7	95.7	95.7	92.8
S5						\	97.0	96.9	94.1	94.1	94.1	94.1	91.2
S7							\	99.9	96.9	96.9	96.9	96.9	93.9
S8								\	96.9	96.9	96.9	96.9	93.9
S9									\	99.4	99.4	99.4	96.2
S10										\	99.9	99.9	96.6
S11											\	99.9	96.7
S12												\	96.8

↑ PPU set — xPPU set ↓

%	S14	S15	S16	S17	S18	S19	S20	S21	S24
S14	\	87.4	83.6	81.5	80.5	79.8	73.2	71.1	70.3
S15		\	94.8	91.9	90.6	89.3	80.7	78.0	77.1
S16			\	96.8	95.4	93.8	84.2	81.3	80.3
S17				\	98.5	96.6	86.4	83.3	82.2
S18					\	97.9	87.4	84.2	83.2
S19						\	89.0	85.7	84.6
S20							\	96.0	94.6
S21								\	98.3

new functionality is introduced (cf. Tab. 2). For the evaluated scenarios (cf. Tab. 2), we conclude that our VAT compares respective systems efficiently. Moreover, we argue that given the runtimes' linear trend (cf. Fig. 7) for the evaluated scenarios, we consider our analysis (cf. Sec. 3.5) to scale even for a large number of comparisons. We consider memory usage vital for VATs applicability and we did not observe a noticeable increase of memory usage during comparison or execution of the VAT. Overall, we consider the VAT feasible to analyze IEC61131-3 control systems. However, inspection of the scenarios revealed further challenges, which we plan to address in the future (cf. Sec. 6). For instance, IEC61131-3 programming languages can *intertwine* directly, i.e., an SFC implementation using ST. Currently, our metric does not cope with nested languages (cf. Tab. 1).

Threats to Validity We assess two case studies from a single domain only. We argue that evaluated scenarios exhibit a considerable complexity and, to some extent, are representative for systems in aPs. We acknowledge that other domains may entail peculiarities we did not consider, thereby impinging results. However, VAT can cope with PLCopen projects in general and, thus, we perceive the VAT to be a promising first step. We further analyze syntactical properties only. Evaluating semantics can be reduced to program equivalence, an undecidable problem. Results produced by the VAT depend on the utilized metric and, thus, on weights and thresholds, on which users may judge differently. We argue that we explicitly provide user with the capability to customize metrics, weights and thresholds to meet their expectations.

5 RELATED WORK

Numerous variability management tools have been proposed, but only few that analyze software systems for their variability [2, 21]. Explicit variability mechanism, such as *if-defs*, are evaluated using the VITAL tool in [29] to extract variability reflexion models. With the VAT, our analysis is not limited to explicit variability mechanism, but it is designed to cope with systems not exhibiting such

information. Analyzing configuration files for the Linux kernel, the *LVAT* tool [5] creates a feature model to represent identified variability. Unlike with the VAT, only explicit variability mechanism are analyzed. Identifying relations between variation points and system variants, the *ISMT4SPL* tool creates variability models. Explicit variability mechanism must be present for *ISMT4SPL*, whereas the VAT aims to first extract variability information. Analyzing electrical circuits, the *VARMA* [19] tool aims to optimize their design and manufacturing process. Unlike with the VAT, identified variability is not classified and no family model is created. Feature models are analyzed using the *FAMA* tool [25], which is limited to feature models, whereas the VAT is designed to analyze multiple different implementation languages. Analyzing variability in system variants, the *MoVa2PL* approach [15] allows to semi-automatically identify their features. Unlike the VAT, the *MoVa2PL* approach does not calculate similarities between individual artifacts based on a customizable metric. With the *pure::variants* tool [1], similarities between system variants are calculated. Unlike with the VAT, calculating similarity values requires a feature selection and, thus, a feature model.

6 CONCLUSION AND FUTURE WORK

For aPs, recent work [7] identified variability management to remain a challenge and tool support to be largely missing.

In this paper, we propose the *Variability Analysis Toolkit* (VAT), a framework that allows for the comparison of PLCopen projects using customizable metrics and the visualization of produced results. Our VAT allows users to compare projects and examine detailed results to the level of individual implementation artifacts. Moreover, we visualize results in the form of a family model. Metrics and produced results are stored and can be exchanged freely between users. Using the PPU, a universal demonstrator for studying evolution in aPs, we demonstrate the VATs to be feasible for the comparison of PLCopen projects. Specifically, we assesses 24 control software system variants implemented in accordance with the IEC61131-3 standard and showed calculated results to be reasonable and performance to be acceptable. Overall, we consider the VAT to be a promising first step towards sophisticated tool support for analyzing variability in automation software systems.

For future work, we plan to extend our metric to remaining IEC61131-3 programming languages. We intent to enrich the VAT with means to cope with intertwined implementations, hence, nested languages. We plan to extend our library of comparison attributes and investigate language-specific traversal algorithms. We intend to conduct survey with industrial partners to assess the VATs applicability in an industrial setting and to get insights on further expectations, i.e., metrics and visualizations. Finally, we aim to compare more than two PLCopen projects, thereby an entire system portfolio.

7 ACKNOWLEDGMENTS

This work has been supported by the DFG (German Research Foundation) (SCHA 1635/12-1) and (VO 937/31-1).

REFERENCES

[1] M. Al-Hajjaji, M. Schulze, and U. Ryssel. 2018. Similarity Analysis of Product-line Variants. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 226–235.

[2] R. Bashroush, M. Garba, R. Rabiser, I. Groher, and G. Botterweck. 2017. CASE Tool Support for Variability Management in Software Product Lines. *ACM Comput. Surv.* 50, 1 (2017), 14:1–14:45.

[3] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz. 2010. Model Clone Detection in Practice. In *Proc. of the Intl. Workshop on Software Clones (IWSC)*. ACM, 57–64.

[4] Z. Durdik, B. Klatt, H. Koziolek, K. Krogmann, J. Stammel, and R. Weiss. 2012. Towards Sustainability Guidelines for long-living Software Systems. In *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. 517–526.

[5] S. El-Sharkawy, A. Krafczyk, and K. Schmid. 2015. Analysing the Kconfig Semantics and Its Analysis Tools. In *Proc. of the Intl. Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 45–54.

[6] N. Ernst, S. Bellomo, I. Ozkaya, R. Nord, and I. Gorton. 2015. Measure it? Manage it? Ignore it? Software Practitioners and Technical Debt. In *Proc. of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 50–60.

[7] J. Fischer, S. Bougouffa, A. Schlie, I. Schaefer, and B. Vogel-Heuser. 2018. A Qualitative Study of Variability Management of Control Software for Industrial Automation Systems. *Proc. of the Intl. Conference on Software Maintenance and Evolution (ICSME)* 13 (2018), 615–624.

[8] H. Grönniger, H. Krahn, C. Pinkernell, and B. Rumpe. 2014. Modeling Variants of Automotive Systems using Views. (2014).

[9] S. Holthusen, D. Wille, C. Legat, S. Beddig, I. Schaefer, and B. Vogel-Heuser. 2014. Family Model Mining for Function Block Diagrams in Automation Software. In *Proc. of the Intl. Workshop on Reverse Variability Engineering (REVE)*. ACM, 36–43.

[10] Q. Huan Dong, F. Ocker, and B. Vogel-Heuser. 2019. 13 (04 2019), 273–282.

[11] International Electrotechnical Commision. 2009. Programmable Logic Controllers – Part 3: Programming Languages.

[12] C. Kolassa, H. Rendel, and B. Rumpe. 2015. Evaluation of Variability Concepts for Simulink in the Automotive Domain. In *Hawaii Intl. Conference on System Sciences (HICSS)*. IEEE, 5373–5382.

[13] C. Legat, J. Folmer, and B. Vogel-Heuser. 2013. Evolution in Industrial Plant Automation: A Case Study. In *Annual Conference of the IEEE Industrial Electronics Society*. 4386–4391.

[14] V. I. Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady* 10, 8 (1966), 707–710.

[15] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. l. Traon. 2015. Automating the Extraction of Model-Based Software Product Lines from Model Variants. In *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. IEEE, 396–406.

[16] C. Riva and C. Del Rosso. 2003. Experiences with Software Product Family Evolution. In *Proc. of the Joint Workshop on Software Evolution and Intl. Workshop on Principles of Software Evolution (IWPSE-EVOL)*. IEEE, 161–169.

[17] J. Rubin and M. Chechik. 2012. Combining Related Products into Product Lines. In *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 285–300.

[18] J. Rubin and M. Chechik. 2013. Quality of Merge-Refactorings for Product Lines. In *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 83–98.

[19] G. Russell, F. Burns, and A. Yakovlev. 2012. VARMA-VARiability Modelling and Analysis Tool. In *Symposium on Design and Diagnostics of Electronic Circuits Systems*. 378–383.

[20] M. Schulze, J. Mauersberger, and D. Beuche. 2013. Functional Safety and Variability: Can it be brought together?. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM.

[21] S. Segura, A. D. Toro, A. B. Sánchez, D. Le Berre, E.l Lonca, and A. R. Cortés. 2015. Automated Metamorphic Testing of Variability Analysis Tools. *Softw. Test., Verif. Reliab.* 25 (2015), 138–163.

[22] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. 2001. The Structure and Value of Modularity in Software Design. In *Proc. of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 99–108.

[23] Technical University in Munich, Germany - Institute of Automation and Information Systems. 2003. The Pick and Place Unit Demonstrator for Evolution in Industrial Plant Automation. http://www.ppu-demonstrator.org

[24] The Instrumentation, Systems, and Automation Society. 1995. Batch Control Part 1: Models and Terminology.

[25] P. Trinidad, D. Benavides, A. Ruiz-CortÃĺs, S. Segura, and A. Jimenez. 2008. FAMA Framework. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. 359–359.

[26] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy. 2015. Evolution of Software in Automated Production Systems: Challenges and Research Directions. *Journal of Systems and Software* 110 (2015), 54 – 84.

[27] B. Vogel-Heuser, T. Simon, and J. Fischer. 2016. Variability Management for Automated Production Systems Using Product Lines and Feature Models. In *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*. 1231–1237.

[28] D. Wille. 2014. Managing Lots of Models: The FaMine Approach. In *Proc. of the Intl. Symposium on the Foundations of Software Engineering (FSE)*. ACM, 817–819.

[29] B. Zhang and M. Becker. 2014. Variability Code Analysis Using the VITAL Tool. In *Proc. of the Intl. Workshop on FOSD*. ACM, 17–22.

Exploring the Variability of Interconnected Product Families with Relational Concept Analysis

Jessie Carbonnel, Marianne Huchard and Clémentine Nebut
firstname.lastname@lirmm.fr
LIRMM, Université de Montpellier, CNRS, Montpellier, France

ABSTRACT

Among the various directions that SPLE promotes, extractive adoption of complex product lines is especially valuable, provided that appropriate approaches are made available. Complex variability can be encoded in different ways, including the feature model (FM) formalism extended with multivalued attributes, UML-like cardinalities, and references connecting separate FMs. In this paper, we address the extraction of variability relationships depicting connections between systems from separate families. Because Formal Concept Analysis provides suitable knowledge structures to represent the variability of a given system family, we explore the relevance of Relational Concept Analysis, an FCA extension to take into account relationships between different families, to tackle this issue. We investigate a method to extract variability information from descriptions representing several inter-connected product families. It aims to be used to assist the design of inter-connected FMs, and to provide recommendations during product selection.

CCS CONCEPTS

• **Information systems → Information extraction**; • **Software and its engineering → Software product lines**; **Software reverse engineering**.

KEYWORDS

Complex Software Product Line, Reverse Engineering, Variability Extraction, Relational Concept Analysis

ACM Reference Format:
Jessie Carbonnel, Marianne Huchard and Clémentine Nebut. 2019. Exploring the Variability of Interconnected Product Families with Relational Concept Analysis. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3307630.3342407

1 INTRODUCTION

As families of similar software systems grow in size and complexity, managing them by adopting an approach based on software product line (SPL) engineering becomes more and more relevant. When various software systems of a same family have been individually developed in an undisciplined way, migrating to an SPL may be done through an extractive approach, by analyzing their commonalities and differences, and by identifying the reusable assets and a reference architecture. Complex variability extraction may result in Feature Models (FMs) [18, 29] extended with multivalued attributes, UML-like cardinalities, or references connecting separate FMs [16].

Previous research work has studied variability extraction in the boolean case [1, 4, 25, 31, 32, 35, 38], and in presence of multi-valued attributes and cardinalities [8, 15]. More specifically, in [15], we show how Formal Concept Analysis (FCA) and its extension to Pattern Structures help in extracting variability relationships from a product family described by multi-valued attributes and cardinalities. It leads to binary implications, groups and mutex involving boolean features as well as attribute values and cardinalities.

In this paper, we address the problem of *cross-family variability relationship extraction* from descriptions representing several inter-connected product families (see Figure 1). These families may correspond to various concerns or product pieces, and may be connected by different relationships, e.g., *use*, *part-of*, *compatible-with*, *provide*. We introduce a method based on Relational Concept Analysis (RCA), an FCA extension to take into account these relationships between the product families. This method produces concept lattice families from which are extracted *cross-family features* and *cross-family relationships* that can be binary implications, mutex, co-occurrences or groups involving both boolean features and cross-family features. This is part of a general approach we envisage, thus we also discuss how in the future this method can assist the design of FMs with new kinds of references, and the exploration of existing configurations for the sake of recommendation.

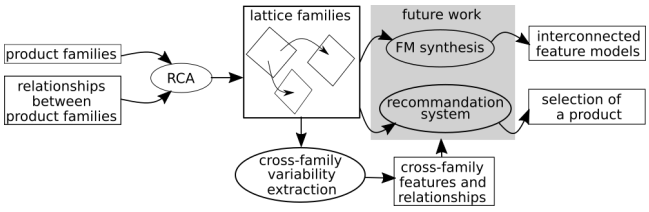


Figure 1: Schema of the envisaged approach.

In Section 2, we introduce the foundations of variability extraction with FCA. Addressing cross-family variability extraction with Relational Concept Analysis is developed in Section 3. Then, we discuss how the extracted cross-family variability can help in interconnected FMs design and in configuration exploration (Section 4). We develop related work in Section 5, before concluding and drawing perspectives of this work in Section 6.

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342407

2 BACKGROUND: ASSISTED VARIABILITY EXTRACTION WITH FCA

Variability modeling. Variability modeling appears in the earliest steps for extractive adoption of a software product line from a family of product variants. To elaborate a variability model in the context of a feature-oriented modeling approach [29], the main existing approaches need to identify (a) representative and discriminating *features* and (b) relationships between these features (e.g. implications, mutually exclusive features or groups). Then the extracted information is represented through *feature models* [18, 29, 37], propositional logics [7], description logics [6], or constraints [36]. Relationships between features are usually extracted from the product variant descriptions depicting variants along with their features. For example, these descriptions may take the form of tabular descriptions, such as Product Comparison Matrices that can be found in Wikipedia[1] and extracted thanks to API such as OpenCompare[2], produced by random or manual sampling from generators, such as using the web application generator JHipster[3][24], or by analysing software developed by communities, such as Robocode[4]. These tabular descriptions generally expose characteristics (as columns) and their values (in cells) for the different product variants (as rows). They need to be cleaned, manually, or automatically [9] before automated exploitation. From the cleaned tabular descriptions, several methods extract relationships for boolean features, leading to FM synthesis or more simply logical relationships extraction [1, 4, 14, 25, 31, 32, 35, 38]. A few approaches [8, 15] address the problem of extracting more complex variability information to take into account multi-valued attributes and cardinalities. Several of these approaches [4, 14, 15, 31, 35, 38] are based on Formal Concept Analysis, which can be seen as a structuring framework for variability analysis and representation, in which some of the other approaches can be embedded, as shown in [3, 14].

Formal Concept Analysis (FCA). Formal Concept Analysis (FCA) [23] can be summarized by the equation: "objects + attributes = concept hierarchy". In other words, based on a set of objects (entities, individuals) described by a set of attributes (properties, characteristics), FCA helps to: (1) group a maximal set of objects sharing a maximal set of attributes into a concept, and (2) hierarchically organize the set of concepts.

In the simplest form, FCA considers as input a *formal context* $K = (O, A, J)$, where O is a set of objects, A is a set of attributes and $J \subseteq O \times A$ is a binary relationship, where $(o, a) \in J$ when "o possesses a". Table 1 represents in a tabular form a formal context KET describing (in a simplified way) Knowledge Engineering (KE) tools (left-hand-side), and a formal context KEC describing basic Knowledge Engineering (KE) components (right-hand-side). The first column of a formal context presents the objects (here in the form of an identifier representing KE tools or components), and the first row gathers the attributes (here the boolean features). KE tools are described by their import/export formats, their distribution mode (commercial or open source) and if they are delivered in SaaS mode (Software as a Service). Basic KE components are dedicated to resource creation (ontology, rules or raw file) or to implement a machine learning algorithm from the categories: rule extraction, decision tree or neural network. The strategy can be supervised or unsupervised. The components can be classified as a symbolic method or as a statistical method.

FCA extracts a set of *formal concepts*. Each concept $C = (E, I)$ is a maximal group of objects E associated with a maximal group of attributes I these objects share. $E = \{o \in O \mid \forall a \in I, (o, a) \in J\}$ is the concept extent and $I = \{a \in A \mid \forall o \in E, (o, a) \in J\}$ is the concept intent. For example (using short names), objects $E_{sy} = \{od1, od2, rd1, f1, re1, re2\}$ share attributes $I_{sy} = \{sy, kec\}$. As these sets cannot be extended, (E_{sy}, I_{sy}) is a concept.

Extent inclusion (and intent containment) induces a specialization order $\leq_{CL}$ between concepts: for two concepts $C_1 = (E_1, I_1)$ and $C_2 = (E_2, I_2)$, $C_1 \leq_{CL} C_2$ if and only if $E_1 \subseteq E_2$ (and equivalently $I_1 \supseteq I_2$). For example, if we consider the concept (E_{rc}, I_{rc}) with $E_{rc} = \{od1, od2, rd1, f1\}$ and $I_{rc} = \{rc, sy, kec\}$, $(E_{rc}, I_{rc}) \leq_{CL} (E_{sy}, I_{sy})$. The *concept lattice* is the set of all concepts C_K of the formal context K, provided with the order $\leq_{CL}$. *Attribute-concepts* (resp. *Object-concepts*) are particular concepts that contain at least an attribute (resp. an object) which is not present in a super-concept (resp. sub-concept). An AOC-poset (for *Attribute-Object-Concept partially ordered set*) is the restriction of the concept lattice to these specific concepts. The AOC-poset which classifies the KE tools (resp. components) of the left-hand-side (resp. right-hand-side) of Table 1 is shown in Figure 2 (resp. Figure 3). A concept is shown as a 3-part box containing: (1) the concept identifier, (2) its intent deprived from the attributes inherited from the super-concepts, and (3) its extent deprived from the objects that appear in the sub-concepts.

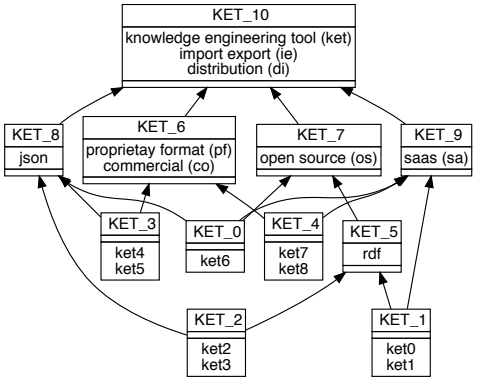


Figure 2: AOC-poset of Knowledge Eng. Tools KET.

Variability information extraction with FCA. Variability information can be extracted from the AOC-poset and used to guide the FM synthesis [14]. Table 2 explains how variability information can be extracted from AOC-posets and translated into FM constructs. The first row considers the situation in which a concept introducing a feature f_2 is a sub-concept of a concept introducing f_1 (denoted by $C_{f_2} <_{CL} C_{f_1}$). In this case, all configurations having f_2 also have f_1, that leads to the propositional formula. Such an implication can be

[1] E.g. comparisons on software systems: https://en.wikipedia.org/wiki/Category: Software_comparisons, last accessed in March 2019
[2] https://github.com/OpenCompare
[3] https://www.jhipster.tech/
[4] https://github.com/but4reuse/RobocodeSPL_teaching

Table 1: Product descriptions of Knowledge Engineering Tools *KET* (lhs), and Knowledge Engineering Components *KEC* (rhs).

KET	knowledge engineering tool (ket)	import export (ie)	rdf	json	proprietay format (pf)	distribution (di)	commercial (co)	open source (os)	saas (sa)
ket0	×	×	×			×		×	×
ket1	×	×	×			×		×	×
ket2	×	×	×	×		×		×	
ket3	×	×	×	×		×		×	
ket4	×	×		×	×	×	×		
ket5	×	×		×	×	×	×		
ket6	×	×		×		×		×	×
ket7	×	×			×	×	×		×
ket8	×	×			×	×	×		×

KEC	knowledge engineering component (kec)	resource creation (rc)	ontology definition (od)	rule definition (rd)	raw file (rf)	learning (le)	algorithm (al)	rule extraction (re)	decision tree (dt)	neural network (nn)	strategy (str)	supervised (su)	unsupervised (un)	symbolic (sy)	statistical (sta)
od1	×	×	×											×	
od2	×	×	×											×	
rd1	×	×		×										×	
f1	×	×			×									×	
re1	×					×	×	×			×	×		×	
re2	×					×	×	×			×		×	×	
dt1	×					×	×		×		×	×			×
nn1	×					×	×			×	×	×			×

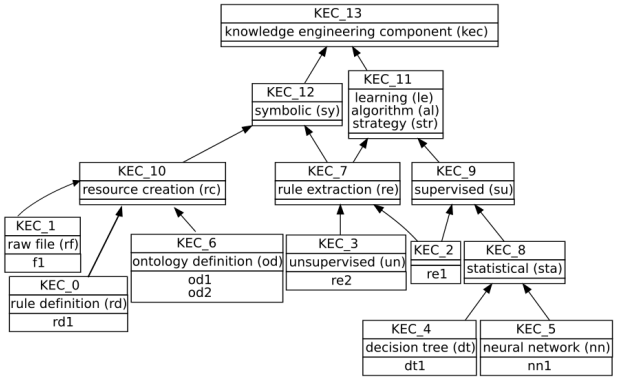


Figure 3: AOC-poset of Knowledge Eng. Components *KEC*.

translated into 3 different FM constructs: a refinement relationship ①, an optional relationship ② or a requires constraint ③.

The second row shows how logical equivalences can be read in AOC-posets. When f_1 and f_2 are introduced in the same concept, this means that features f_1 and f_2 are always present together ($f_1 \leftrightarrow f_2$). In a feature diagram, this can be represented: with a mandatory relationship ④ or with two requires constraints ⑤.

The third row shows how mutual exclusions can be read in an AOC-poset. Two features f_1 and f_2 (respectively introduced in C_{f_1} and C_{f_2}) are mutually exclusive if the common sub-concepts of C_{f_1} and C_{f_2} (denoted $C_{f_1} \sqcap C_{f_2}$) do not introduce any object.

The fourth row presents the mapping of or-groups ⑦ into AOC-posets. We consider $\{f_1, ..f_k\}$ the features involved in an or-group, and f_0 the parent-feature of this group. All configurations having one of the features in $\{f_1, ..f_k\}$ should also have f_0, and conversely, configurations having the parent-feature f_0 have at least one feature of the or-group. Thus, the union of the extents of concepts introducing features of $\{f_1, ..f_k\}$ is equal to the extent of the concept introducing f_0. Moreover, the concept introducing the parent-feature of an or-group is not an object-concept, as at least one feature of $\{f_1, ..f_k\}$ has to be selected. It is denoted by $C_{f_0} \notin \boldsymbol{OC}$, C_{f_0} being the concept introducing the parent-feature, and $\boldsymbol{OC}$ the set of object-concepts of the AOC-poset. Besides, we consider that there always exists a root feature, which appears in the top concept containing all configurations.

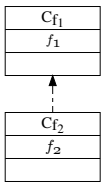
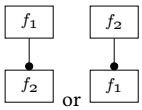
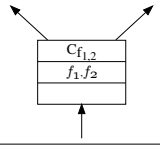
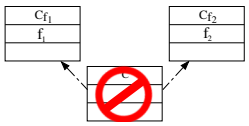
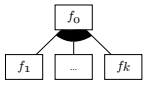
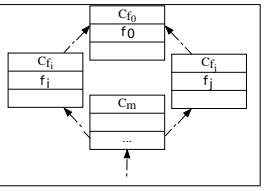
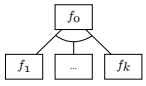
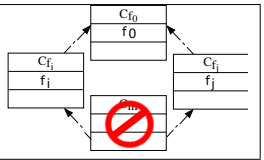
The last row presents the mapping of xor-groups ⑧ into AOC-posets. Xor-groups are like or-groups, but concepts introducing the features of $\{f_1, ..f_k\}$ do not have a common sub-concept introducing an object. Indeed, features involved in a xor-group and features involved in an *exclude* cross-tree constraint have a similar behaviour, as they are mutually exclusive in both situations.

3 CROSS-PRODUCT FAMILY VARIABILITY EXTRACTION WITH RCA

Product line engineering faces inescapable issues related to the growth in size and complexity of software systems and to their evolution [27]. Decomposing systems and considering separated concerns or gradual construction are solutions that have been investigated, e.g. in multi-product lines [10, 27, 33]. Extending the variability extraction from a product family to a set of interconnected product families means that we need to consider both intra-family and inter-family (cross-family) variability information.

Relational Concept Analysis. Cross-family variability extraction can be assisted by Relational Concept Analysis (RCA) [34] which considers several object categories (one per product family). While FCA groups objects sharing commonalities in their intrinsic attributes, RCA additionally groups objects sharing commonalities in their similar relations to other objects having themselves commonalities. To that aim, RCA iteratively applies FCA to propagate similarities from objects in one category to objects in (possibly)

Table 2: Mapping between AOC-posets and FMs [14]. The extent of a context C is denoted by $EXT(C)$

FMs	Prop. form.	AOC-posets	
①  ②  ③ $f_2 \rightarrow f_1$	$f_2 \rightarrow f_1$	$C_{f_2} \leq_{CL} C_{f_1}$	
④  ⑤ $f_1 \rightarrow f_2$ $f_2 \rightarrow f_1$	$f_1 \leftrightarrow f_2$	$C_{f_1} =_{CL} C_{f_2}$	
⑥ $f_1 \rightarrow \neg f_2$	$f_1 \rightarrow \neg f_2$ or $f_2 \rightarrow \neg f_1$	$Ext(C_{f_1} \sqcap C_{f_2}) = \varnothing$	
⑦ 	$f_0 \leftrightarrow (f_1 \vee \ldots \vee f_k)$	$\forall f_i \in \{f_1, ..f_k\}, C_{f_i} \leq_{CL} C_{f_0}$. $\{C_{f_1}, ...C_{f_k}\}$ is the greatest antichain [28] verifying: $Ext(C_{f_1}) \cup ... \cup Ext(C_{f_k}) = Ext(C_{f_0})$. $C_{f_0} \notin OC$	
⑧ 	$f_0 \leftrightarrow (f_1 \oplus \ldots \oplus f_k)$	$\forall f_i \in \{f_1, ..f_k\}, C_{f_i} \leq_{CL} C_{f_0}$. $\forall f_i, f_j \in \{f_1, ..f_k\} \mid f_i \neq f_j, Ext(C_{f_i} \sqcap C_{f_j}) = \varnothing$. $\{Ext(C_{f_1}), ..., Ext(C_{f_k})\}$ is a partition of $Ext(C_{f_0})$. $C_{f_0} \notin OC$	

another category through relations. To follow up our illustrative example, identifying the group of statistical KE components can be propagated to KE tools through a *provides* relation, leading to identify the group of KE tools dedicated to statistical analyses.

The considered dataset in RCA is a Relational Context Family (RCF), which is a set of object-attribute contexts (objects described by intrinsic attributes) and a set of object-object contexts (relations between objects of the different categories). For our example, the RCF is composed of the object-attribute contexts KEC and KET (Table 1), and the object-object relation $provides \subseteq KET \times KEC$ (Table 4). In the general case, an RCF contains n object-attribute contexts $K_i = (O_i, A_i, J_i), i \in \{1, ..., n\}$ and m object-object contexts $R_j = (O_k, O_l, r_j), j \in \{1, ..., m\}$, with $r_j \subseteq O_k \times O_l$ is a binary relation such that $k,\ l \in \{1, ..., n\}$, O_k is the domain of the relation, and O_l is the range of the relation.

To consider the *provides* relation, information is added to the description of objects of its domain (KET). A straightforward integration scheme could be to simply add to the object-attribute context KET attributes of the form $(provides, kec)$, and to add a cross when an object of KET provides the corresponding object of KEC. For example, $(provides, od1)$ could be added to the description of $ket0$ in a new column of KET. This would allow to group KETs that own the same KECs, but we would loose the valuable knowledge given by the KEC concepts. Remember that these concepts indicate what are the shared characteristics of KECs, e.g. $dt1$ and $nn1$ are grouped in KEC_8 because both are statistical-based KECs (see Fig. 3). Now if we consider the way *provides* describes $ket7$ and $ket8$ (resp. by $(provides, dt1)$ and $(provides, nn1)$), these two KETs do not share any KEC, whereas they share the fact that they allow statistical-based learning. To acquire such information, we propagate the knowledge highlighted in the KEC AOC-poset to the KET AOC-poset through *relational attributes*. These relational attributes are formed using scaling quantifiers inspired by constructors used in descriptions logics [5], such as $\exists$ and $\exists\forall$[5], as illustrated in Table 3. For example, $ket7$ and $ket8$ descriptions will be enhanced by a relational attribute $\exists provides(KEC\_8)$ to indicate that both $ket7$ and $ket8$ have a link to at least one object of the KEC_8 extent.

In the RCA process, a scaling quantifier q is associated with each relation r, and systematically applied to form all the relational attributes of the form $qr(C)$ with C a concept formed on the objects of the range of r. These relational attributes are added to the context of the domain of r and a new extended AOC-poset can be built. For

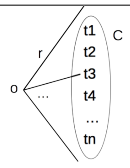
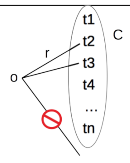
[5] $\exists\forall$ is used instead of $\forall$ to avoid assigning to an object o a relational attribute formed on r when $r(o)$ is empty (and would be included in any concept extent).

example, Fig. 4 shows the AOC-poset built for *KET* extended with the relational attributes formed with the quantifier ∃∀ applied to *provides*. In this AOC-poset, *KET*_4 now highlights new shared information about *ket*7 and *ket*8 which is the fact that both provide **only** (∃∀) statistical-based learning components (the components they provide are all in *KEC*_8 extent).

RCA can consider complex and possibly cyclic entity-relationship models. A non-trivial modeling phase is central in this perspective, to determine which are the object-attribute contexts (separate product families) and which are the object-object contexts (interconnections between the families), and the adequate scaling quantifiers.

Cross-family features. Table 3 shows two examples of the new features that this framework makes available to describe cross-family variability. These features are abstractions of links between an object *o* (a product configuration of the domain of *r* in our context) and a concept *C* (a group of product configurations of the range of *r*). Existential features (∃*r*(*C*)) indicate if a configuration *o* is linked by *r* to at least one configuration in the extent of Concept *C*. In our example, ∃*provides*(*C*) is the feature "provides a KEC with the features of C". Feature ∃*provides*(*KEC*_8) can be read "provides a statistical learning component". Universal features (∃∀*r*(*C*)) indicate if all *r* links of a configuration *o* are towards configurations in the extent of Concept *C* ; for our example, it can be summarized as "provides only KEC with the features of C". As presented previously, ∃∀*provides*(*KEC*_8) can be read "provides only statistical learning components".

Table 3: Examples of cross-family features

Object links vs. concept extent	Schema	Relational attribute Cross-family Feature
$r(o) \cap Extent(C) \neq \emptyset$		existential $\exists r(C)$
$r(o) \subseteq Extent(C)$ $r(o) \neq \emptyset$		universal $\exists \forall r(C)$

Cross-family variability relationships. Variability information can be extracted from the AOC-poset resulting from RCA, using the same mappings as introduced in Table 2. The found relationships also involve cross-family features. Three examples are given below using the AOC-poset of Figure 4.

Example of co-occurrence. By the second row of Table 2, rdf and ∃∀*provides*(*KEC*_12) are co-occurrent, since both are introduced in the same *KET*_5. In other words, since *KEC*_12 corresponds to symbolic components, that means that tools exporting in rdf only use symbolic methods.

Example of implication. By the first row of Table 2, *KET*_1 is below *KET*_9, thus ∃∀*provides*(*KEC*_6) → *Saas*: the knowledge engineering tools providing components that only deal with ontology definition (*KEC*_6) are delivered in SaaS mode. This can be translated through several ways in a feature model (see cases ①, ② and ③ in Table 2).

Example of exclusion. *KET*_5 and *KET*_6 do not have a common sub-concept introducing a configuration. By the third row of Table 2, ∃∀*provides*(*KEC*_6) → ¬pf. We can deduce that none of the KETs providing components based only on symbolic methods (*KEC*_12) is based on a proprietary format.

4 EXPLOITING CROSS-FAMILY VARIABILITY

In this section, we draw tracks of future research to exploit cross-family variability information.

Extended (semi-automated) FM synthesis. Intra-family information has been used to assist FM synthesis [1, 3, 15, 18, 25, 26, 35, 39, 40]. The methods that use FCA to assist FM synthesis [3, 15, 35] exploit a concept structure to derive automatically an FM or to provide user guidance by reducing their choices during FM construction. For example, the FM of KETs in Fig. 5 (resp. of KECs in Fig. 6) can be extracted from the AOC-poset of Fig. 2 (resp. Fig. 3) by an FM designer, assisted by the rules presented in Sect. 2. Cross-family variability information can in turn be used to assist the synthesis of interconnected FMs, either by writing constraints between two different FMs (as shown in Sect. 3), or to enhance an FM from one family by features coming from knowledge about relations in real configurations. We illustrate this by using information extracted from the AOC-poset of *KET* (Fig. 4). First, we introduce a reference `kec` below the FM root, with cardinality *1-many (*)*. Then the AOC-poset states that in observed real configurations (as shown in *KET*_4), some KETs propose only supervised learning algorithms (∃∀*provides*(*KEC*_9)), thus if a user chooses that option when he designs a KET, he should be guided afterwards to configure only components implementing supervised learning algorithms. The AOC-poset also shows (in *KET*_5) that a group of KETs is dedicated to symbolic approaches only (∃∀*provides*(*KEC*_12)) and a subgroup of them is dedicated to ontology definitions only (∃∀*provides*(*KEC*_6)). From these observations, the FM designer could decide to introduce a feature `supervised ket (sup)`, a feature `symbolic ket (sy)` with a sub-feature `ontology dedicated (o)`. These choices are represented as an extension of the FM *KET*, as shown in Fig. 7 in the grey rectangular box. The introduced features in the KET FM should be used in configuration tools to reduce the choices in the KEC FM when they are selected. It can be tricky for an FM designer to choose to add features to the KET FM depending on the relations that are observed between KET configurations and KEC configurations. This indeed propagates structuring information from the target of *provides* (KEC) to the source (KET), and in many cases, this could break the separation of concerns. In this case, it should be preferred to use cross-family constraints. Using cross-family constraints to write cross-tree constraints nevertheless requires that the concept carrying the constraint corresponds to a feature. If this is not the case the constraint should be redistributed on the existing children if any exists.

Introducing in a feature model FM_1 cross-family features that are derived from another feature model FM_2 can be seen as a way to achieve feature model composition, and especially the union

provides	od1	od2	rd1	f1	re1	re2	dt1	nn1
ket0	×							
ket1	×	×						
ket2	×	×			×			
ket3		×	×			×		
ket4	×	×					×	×
ket5				×	×	×	×	×
ket6				×				×
ket7							×	
ket8								×

Table 4: Relation *provides* between Knowledge Engineering Tools *KET* (rows) and Knowledge Engineering Components *KEC* (columns).

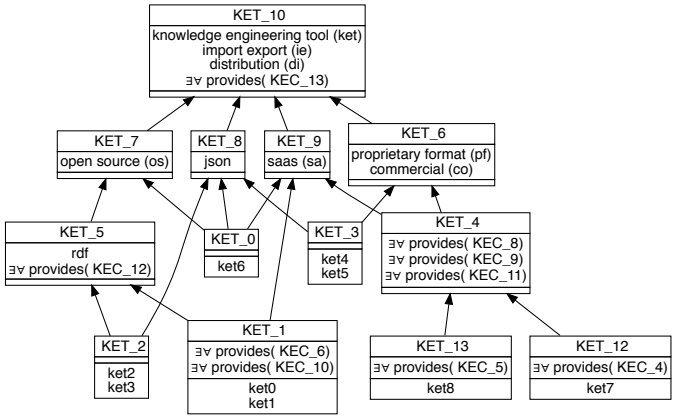


Figure 4: AOC-poset of Knowledge Engineering Tools *KET* after integration of information about *provides*, using ∃∀ quantifier.

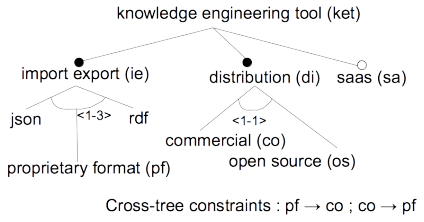


Figure 5: an FM extracted from the AOC-poset of Figure 2.

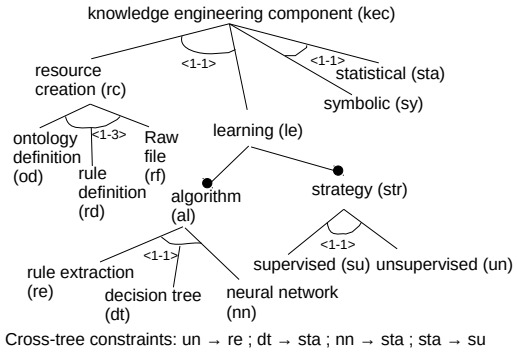


Figure 6: an FM extracted from the AOC-poset of Figure 3.

operation. Traditional methods merge the common features of two models and keep the specific features of both initial FMs. It produces a single output FM gathering features from two different concerns into one model [2, 13]. Cross-family features group features of interest from FM_2 into more "abstract features" that will enrich FM_1 with external concerns.

Recommendation. Information embodied in AOC-posets can be used to guide the users in their choices when selecting a valid configuration. Concepts represent maximal groups of valid configurations sharing a maximal set of features, that may be intrinsic or

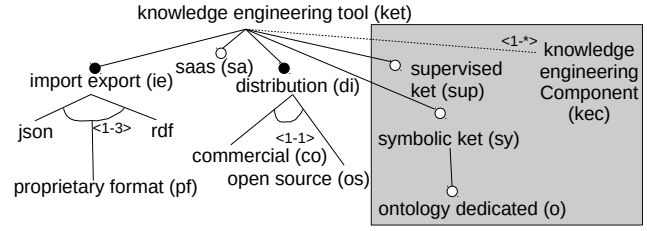


Figure 7: FM that can be extracted from the AOC-poset of Figure 4, and cross-family information.

cross-family. If we consider that the conjunction of a concept's features forms a query, then the concept's set of configurations can be seen as the result of this query. The concepts kept in an AOC-poset represent the maximal queries that can be formulated by a user, i.e., the maximal feature conjunctions representing each configuration subset. For instance, if a user wants to retrieve all KETs supporting the RDF format, the corresponding maximal query in Figure 4 corresponds to the concept introducing the feature rdf (KET_5), and the result is $ket0$-3. This can be applied for cross-family features too: it enables to query the KET configurations depending on their relationships to KECs. For instance, if a user wants to retrieve KETs providing only neural network KECs (KEC_5), this query corresponds to the concept KET_13 of Figure 4 introducing $\exists\forall provides(KEC\_5)$ and it results to the unique configuration $ket8$. The querying can benefit from the variety of quantifiers and their different semantics [11]. If, this time, a user wants to retrieve KETs providing at least one neural network KEC, we should consider the AOC-poset built with the existential quantifier.

The different quantifiers are ordered by generalization [11] thus a concept formed in an AOC-poset with a quantifier q_s can be projected in the AOC-poset formed with q_g, if q_g is more general than q_s. For instance, $\exists$ is more general than $\exists\forall$. Choosing the right quantifier can be useful to tune the degree of observed variability and the precision of the recommendation systems.

Concepts' position in the AOC-poset and to each other reveals other useful information about the structured family. Concepts on top of the AOC-poset usually possess less features and more configurations and therefore represent groups of features shared by most of the considered configurations. Dually, concepts at the bottom generally possess more features and less configurations, and represent more specialised features, i.e., the ones shared by few configurations. For instance Figure 3 reveals that there are more symbolic KECs (KEC_12) than statistical ones (KEC_8): this information may be used in recommendation systems to propose, for instance, popular features to a user. It is also noteworthy that the closer two concepts are in the structure, the more similar are their sets of features and configurations. Therefore, navigating from one concept to one of its neighbours represents modifications that can be applied on the corresponding query. This navigation has different properties depending the used conceptual structure (e.g. AOC-poset, concept lattice). Retrieving a concept based on a set of features can be seen as *querying* information from the relation concept family. Navigating from a concept to another in the structure enables a user to *explore* the set of available configurations by progressively refining the query. Let us imagine that a user wants to retrieve all the available open source KETs: this query corresponds to KET_7 of Figure 4. If the user wants to refine this query to find a more specific KET, the AOC-poset can be used to recommend the smallest modifications that may be applied: here, it proposes to either explore KETs having features *json* and *saas* (i.e., moving to KET_0) or explore KETs supporting rdf format and providing only symbolic KECs (i.e., moving to KET_5). Cross-family features allow to switch between AOC-posets: here, the user can "jump" to the concept referenced by $\exists\forall provides(KEC\_12)$ in the KECs' AOC-poset of Figure 3 and then refine the corresponding query. For instance, they can choose to focus on *symbolic* KECs specialised on *ressource creation* (KEC_10) and then jump back to the previous KETs' AOC-poset, but this time in the concept introducing $\exists\forall provides(KEC\_10)$ (i.e., KET_1) with $ket0$ and $ket1$.

5 RELATED WORK

Interconnected variability models. Several tracks have been followed to represent modularity, and FMs extensions have been proposed, among which we can notice *Feature Models with References* (FMR) [16, 17] and *Modular Feature Models* (MFM) [6]. These two extensions enable the separation of a single FM into several FMs dedicated to sub-product lines or to separate concerns. In FMRs, a reference is a feature of an FM representing the root of another FM. Cross-FM constraints may be established between features of the two FMs. In MFMs, the authors encourage the definition of FMs modules, and the features in different FM modules are connected through cross-FM implications in *module bridges*. The approach is developped using the description logic $\mathscr{ALCH}$ for describing the FMs and the cross-tree and cross-FM constraints.

Friess et al. [21] work on modelling composition rules between different feature models representing independent product lines that can be combined. The authors define what they call a *feature configuration*, which represents a subset of an FM valid configurations satisfying some constraints. Then, they define composition rules (e.g., *uses*, *parts-of*) between feature configurations of different FMs. Feature configurations permit to finely tune the set of configurations involved in a composition rule. Similarly, Rosenmuller et al. [33] work on reinforcing constraint expressiveness between different yet connected FMs by relying on *product line specialisation* [17]. The latter was introduced by Czarnecki et al. [17] as a way to prune the set of valid configurations of an FM by partially configuring this FM. Rosenmuller et al. argue that, when modelling complex product line, relying on domain constraints at the domain level is not sufficient, and that modelling constraints involving product line instances (i.e., valid configurations) is necessary. They use UML class diagrams to model these constraints on configurations, where a class represents an FM, a sub-class represents an FM specialisation and relations are used to define constraints at the configuration level. These "instance models" are used in association with classic domain models such as FMs. Urli et al. [41] propose a domain model regrouping several FMs and relations between these FMs. In this approach, the domain model defines the different abstract concepts of the modelled complex product line, and each FM characterises the variability of one of these concepts. Contrarily to approaches presented before, constraints between FMs involved features and not subsets of configurations. Dhungana et al. [20] propose an approach to ease the integration of different kinds of variability models into a unique infrastructure. They present `Invar`, a framework allowing to manage a repository of different variability models. When a new model is loaded , constraints between the new model and the existing ones need to be specified. These constraints take the form of "if *condition* then *action*". Conditions involve the selection or deselection of a feature. Actions may be to include another variability model (similar to model modularisation of [17]), or to select or deselect a feature in another variability model.

Extraction of interconnected variability models. Most of the existing methods for FM reverse engineering exclusively focus on boolean FMs [1, 18, 19, 26, 30, 35]. Becan et al [8] were the first to propose a reverse engineering method for more complex FMs taking the form of FMs with attributes. During previous work [15], we introduced the usage of Pattern Structures [22] to extract complex variability information in the form of logical relationships corresponding to the logical semantics of FMs extended with both multivalued attributes and UML-like cardinalities. Here we elaborate on RCA, both being possibly combined. To the best of our knowledge, there is still no work about the extraction of "cross-family" variability information.

6 CONCLUSION

As software systems grow and the software product line engineering is spreading, collaborative design of huge product lines which combine several concerns will become more and more critical. Complex variability can take various forms, including variability among inter-connected software families. In this paper, we address the aim to extract such *cross-family variability* from a set of interconnected product configurations. We propose to employ Relational Concept Analysis (RCA), an extension of Formal Concept Analysis to assist this extraction. We introduce *cross-family features* and *cross-family relationships* that take the form of binary implications, mutex, co-occurrences or groups that involve both boolean features and cross-family features. Extracting such information has

the inherent complexity of related data-mining methods. We expect reasonable complexity for exact extraction of binary implications, mutex, co-occurrences, and probably difficulties, if not unfeasibility, for groups, requiring approximate methods. We then discuss the possibility to use this variability information to assist the design of FMs with new kinds of references, or to explore a set of inter-connected configurations. As future work, we plan to apply the method to inter-connected product descriptions, such as connected PCMs, or by dividing large tabular descriptions, like the one produced for JHipster in [24], into separate concerns. Previous work that applied FCA and RCA to Wikipedia's or synthetic PCMs [12] gave us some preliminary results on the feasibility. Exploring the effects of the different RCA scaling quantifiers on these datasets will be very informative to tune the method. We also are interested by the possibilities of composing different inter-connected FMs with the support of our method. Finally, we will study the way this cross-family variability has to be taken into account to guide the users during a product selection/construction.

REFERENCES

[1] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. 2012. On extracting feature models from product descriptions. In *Proc. of the 6th Int. Works. on Variability Modelling of Soft.-Intensive Syst. (VaMoS'12)*. 45–54.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2010. Comparing Approaches to Implement Feature Model Composition. In *Proc. of the 6th Europ. Conf. on Modelling Foundations and Applications (ECMFA'10)*. 3–19.

[3] Ra'Fat Al-Msie 'deen, Abdelhak-Djamel Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Ahmad Al-Khlifat. 2014. Concept lattices: A representation space to structure Soft. variability. In *Proc. of the 5th Int. Conf. on Information and Communication Syst. (ICICS'14)*. 1–6.

[4] Ra'Fat Al-Msie'deen, Marianne Huchard, Abdelhak Seriai, Christelle Urtado, and Sylvain Vauttier. 2014. Reverse Eng. Feature Models from Soft. Configurations using Formal Concept Analysis. In *Proc. of the 11th Int. Conf. on Concept Lattices and Their Applications (CLA'14)*. 95–106.

[5] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider (Eds.). 2003. *The Description Logic Handbook*. Cambridge Univ. Press, Cambridge, UK.

[6] Ebrahim Bagheri, Faezeh Ensan, Dragan Gasevic, and Marko Boskovic. 2011. Modular Feature Models: Representation and Configuration. *Journal of Research and Practice in Information Technology* 43, 2 (2011), 109–140.

[7] Don S. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. of the 9th Int. Soft. Product Line Conf. (SPLC'05)*. 7–20.

[8] Guillaume Bécan, Razieh Behjati, Arnaud Gotlieb, and Mathieu Acher. 2015. Synthesis of attributed feature models from product descriptions. In *Proc. of the 19th Int. Conf. on Soft. Product Line (SPLC'15)*. 1–10.

[9] Guillaume Bécan, Nicolas Sannier, Mathieu Acher, Olivier Barais, Arnaud Blouin, and Benoit Baudry. 2014. Automating the formalization of product comparison matrices. In *Proc. of the ACM/IEEE Int. Conf. on Aut. Soft. Eng., (ASE'14)*. 433–444.

[10] Goetz Botterweck. 2013. Variability and Evolution in Systems of Systems. In *Proc. of the 1st Workshop on Advances in Systems of Systems (AiSoS'13)*. 8–23.

[11] Agnès Braud, Xavier Dolques, Marianne Huchard, and Florence Le Ber. 2018. Generalization effect of quantifiers in a classification based on relational concept analysis. *Knowl.-Based Syst.* 160 (2018), 119–135.

[12] Jessie Carbonnel, Marianne Huchard, and Alain Gutierrez. 2015. Variability Representation in Product Lines using Concept Lattices: Feasibility Study with Descriptions from Wikipedia's Product Comparison Matrices. In *Proc. of ws. FCA&A 2015, co-loc. 13th Int. Conf. on Formal Concept Analysis (ICFCA)*. 93–108.

[13] Jessie Carbonnel, Marianne Huchard, André Miralles, and Clémentine Nebut. 2017. Feature Model Composition Assisted by Formal Concept Analysis. In *Proc. of the 12th Int. Conf. on Evaluation of Novel App. to Soft. Eng. (ENASE'17)*. 27–37.

[14] Jessie Carbonnel, Marianne Huchard, and Clémentine Nebut. 2019. Modelling equivalence classes of feature models with concept lattices to assist their extraction from product descriptions. *Journ. of Syst. and Soft.* 152 (2019), 1 – 23.

[15] Jessie Carbonnel, Marianne Huchard, and Clémentine Nebut. 2019. Towards complex product line variability modelling: Mining relationships from non-boolean descriptions. *Journ. of Syst. and Soft. (doi:10.1016/j.jss.2019.06.002)* (2019).

[16] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich W. Eisenecker. 2002. Generative Programming for Embedded Soft.: An Industrial Experience Report. In *Proc. of the 1st ACM SIGPLAN/SIGSOFT Conf. on Generative Programming and Component Eng. (GPCE'02)*. 156–172.

[17] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. 2004. Staged Configuration Using Feature Models. In *Proc. of the 3rd Int. Soft. Product Line Conf. (SPLC'04)*. 266–283.

[18] Krzysztof Czarnecki and Andrzej Wasowski. 2007. Feature Diagrams and Logics: There and Back Again. In *Proc. of the 11th Int. Soft. Product Line Conf. (SPLC'07)*. 23–34.

[19] Jean-Marc Davril, Edouard Delfosse, Negar Hariri, Mathieu Acher, Jane Cleland-Huang, and Patrick Heymans. 2013. Feature model extraction from large collections of informal product descriptions. In *Proc. of the 9th Joint Meeting of the Europ. Soft. Eng. Conf. and the ACM SIGSOFT Symposium on the Foundations of Soft. Eng. (ESEC/FSE'13)*. 290–300.

[20] Deepak Dhungana, Dominik Seichter, Goetz Botterweck, Rick Rabiser, Paul Grünbacher, David Benavides, and José A. Galindo. 2011. Configuration of Multi Product Lines by Bridging Heterogeneous Variability Modeling App.. In *Soft. Product Lines - 15th Int. Conf., SPLC 2011*. 120–129.

[21] Wolfgang Friess, Julio Sincero, and Wolfgang Schroeder-Preikschat. 2007. Modelling compositions of modular embedded Soft. product lines. In *Proc. of the 25th Conf. on IASTED Int. Multi-Conf.: Soft. Eng.* ACTA Press, 224–228.

[22] Bernhard Ganter and Sergei O. Kuznetsov. 2001. Pattern Struct. and Their Projections. In *Proc. of the 9th Int. Conf. on Conceptual Struct. (ICCS'01)*. 129–142.

[23] Bernhard Ganter and Rudolf Wille. 1999. *Formal concept analysis - mathematical foundations*. Springer.

[24] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Patrick Heymans. 2017. Yo Variability! JHipster: A Playground for Web-Apps Analyses. In *11th Int. Works. on Variability Modelling of Soft.-intensive Syst.* Eindhoven, Netherlands, 44 – 51. https://doi.org/10.1145/3023956.3023963

[25] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2011. Reverse Eng. Feature Models from Programs' Feature Sets. In *Proc. of the 18th Work. Conf. on Reverse Eng. (WCRE'11)*. 308–312.

[26] Evelyn Nicole Haslinger, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2013. On Extracting Feature Models from Sets of Valid Feature Combinations. In *Proc. of the 16th Int. Conf. on Fundamental App. to Soft. Eng. (FASE'13)*. 53–67.

[27] Gerald Holl, Paul Grünbacher, and Rick Rabiser. 2012. A Systematic review and an expert survey on capabilities supporting multi product lines. *Information & Soft. Technology* 54, 8 (2012), 828–852.

[28] Guy-Vincent Jourdan, Jean-Xavier Rampon, and Claude Jard. 1994. Computing on-line the lattice of maximal antichains of posets. *Order* 11, 3 (1994), 197–210.

[29] 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. Soft. Eng. Institute.

[30] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Feature Model Synthesis with Genetic Programming. In *Proc. of the 6th Int. Symposium on Search-Based Soft. Eng. (SSBSE'14)*. 153–167.

[31] Felix Loesch and Erhard Ploedereder. 2007. Restructuring Variability in Soft. Product Lines using Concept Analysis of Product Configurations. In *Proc. of the 11th Europ. Conf. on Soft. Maintenance and ReEng. (CSMR'07)*. 159–170.

[32] Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2017. Bottom-up technologies for reuse: Automated extractive adoption of Soft. product lines. In *Comp. Proc. of the 39th Int. Conf. on Soft. Eng. (ICSE'17)*. 67–70.

[33] Marko Rosenmüller, Norbert Siegmund, Christian Kästner, and Syed Saif ur Rahman. 2008. Modeling dependent software product lines. In *Proc. of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE'08)*. 13–18.

[34] Mohamed Rouane-Hacene, Marianne Huchard, Amedeo Napoli, and Petko Valtchev. 2013. Relational concept analysis: mining concept lattices from multi-relational data. *Annals of Math. and Artificial Intelligence* 67, 1 (2013), 81–108.

[35] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. 2011. Extraction of feature models from formal contexts. In *Works. Proceedings (Volume 2) of the 15th Int. Conf. on Soft. Product Lines (SPLC'11)*. 4:1–4:8.

[36] Camille Salinesi, Olfa Djebbi, Raúl Mazo, Daniel Diaz, and Alberto Lora-Michiels. 2011. Constraints: The core of product line Eng.. In *Proc. of the Fifth IEEE Int. Conf. on Research Challenges in Information Science (RCIS'11)*. 1–10.

[37] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic semantics of feature diagrams. *Computer Networks* 51, 2 (2007), 456–479.

[38] Anas Shatnawi, Abdelhak-Djamel Seriai, and Houari A. Sahraoui. 2017. Recovering Soft. product line architecture of a family of object-oriented product variants. *Journal of Syst. and Soft.* 131 (2017), 325–346.

[39] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2011. Reverse Eng. feature models. In *Proc. of the 33rd Int. Conf. on Soft. Eng., (ICSE'11)*. 461–470.

[40] Steven She, Uwe Ryssel, Nele Andersen, Andrzej Wasowski, and Krzysztof Czarnecki. 2014. Efficient synthesis of feature models. *Information & Soft. Technology* 56, 9 (2014), 1122–1143.

[41] Simon Urli, Mireille Blay-Fornarino, and Philippe Collet. 2014. Handling complex configurations in Software product lines: a tooled approach. In *18th Int. Soft. Product Line Conf. (SPLC '14)*. 112–121.

Ontology-Based Security Tool for Critical Cyber-Physical Systems

Abdelkader Magdy Shaaban
Austrian Institute of Technology
Center for Digital Safety & Security
Vienna, Austria
Abdelkader.Shaaban@ait.ac.at

Thomas Gruber
Austrian Institute of Technology
Center for Digital Safety & Security
Vienna, Austria
Thomas.Gruber@ait.ac.at

Christoph Schmittner
Austrian Institute of Technology
Center for Digital Safety & Security
Vienna, Austria
Christoph.Schmittner@ait.ac.at

ABSTRACT

Industry 4.0 considers as a new advancement concept of the industrial revolution, which introduces a full utilization of Internet technologies. This concept aims to combine diverse technological resources into the industry field, which enables the communication between two worlds: the physical and the cyber one. Cyber-physical Systems are one of the special forces that integrate and build a variety of existing technologies and components. The diversity of components and technologies creates new security threats that can exploit vulnerabilities to attack a critical system. This work introduces an ontology-based security tool-chain able to be integrated with the initial stages of the development process of critical systems. The tool detects the potential threats, and apply the suitable security requirements which can address these threats. Eventually, it uses the ontology approach to ensure that the security requirements are fulfilled.

KEYWORDS

Cyber-physical System, Security, Threats, Ontology

ACM Reference format:
Abdelkader Magdy Shaaban, Thomas Gruber, and Christoph Schmittner. 2019. Ontology-Based Security Tool for Critical Cyber-Physical Systems. In *Proceedings of 23rd International Systems and Software Product Line Conference - Volume B, Paris, France, September 9–13, 2019 (SPLC '19),* 3 pages.
https://doi.org/10.1145/3307630.3342397

1 INTRODUCTION

In recent years, industrial production has changed towards a smart manufacturing form based on the integration of both the Internet of Things (IoT) and the Cyber-Physical Systems (CPS). That introduces a new industrial revolution is called Industry 4.0. CPS integrates and builds on a variety of existing technologies and components such as robotics, industrial automation, big data, and cloud computing [1] [2].

The insecure communication protocols and outdated components generate new security issues which can jeopardize the data by different types of attacks. The security aims to protect computer systems and information from various illegal acts [2]. Furthermore, to build a secure system is important to:

- define the exact potential threats that threaten the system,
- identify the most suitable security requirements able to address the identified threats s and reduce the overall risk,
- verify and validate that the security requirements meet the actual security need,

This work introduces a security tool-chain based on the ontology approach to define the security weaknesses in critical CPS applications. It uses a systematic and standard approach, such as an information security management system ISMS [3] and IEC 62443 series [4], which provide a standardized methodology for building a secure infrastructure.

2 THE STRUCTURE OF THE ONTOLOGY-BASED SECURITY TOOL

The security issues in CPS can be generated from the diversity of interconnected components or the existence of legacy units in system architecture. To address the security weaknesses in a system is needed to manage a massive quantity of security requirements to build a secure infrastructure according to a systematic and standard approach. Figure 1 shows a simple Smart Factory application as a good example of CPS application. This example is defined in separate layers; each layer contains different components as follows:

Field Devices: this layer contains sensors for collecting data and actuators as robotic arms for welding, handling, painting, drilling, or other specific function.

Processing Units: this consists of smart devices to process the sensors data and send the suitable action to actuators.

Communication Units: this layer consists of communication devices such as gateways to transfer data over the network of the Smart Factory.

Cloud Storage: this consists of physical cloud storage to store data over the cloud.

Observation: this layer monitors products over long-run to compare production and conditions across the years.

The example contains one legacy gateway (Gateway2) and some insecure communication protocols. The security tool-chain is applied to this example to find potential threats, define the proper security requirements, and ensure that these requirements are fulfilled.

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342397

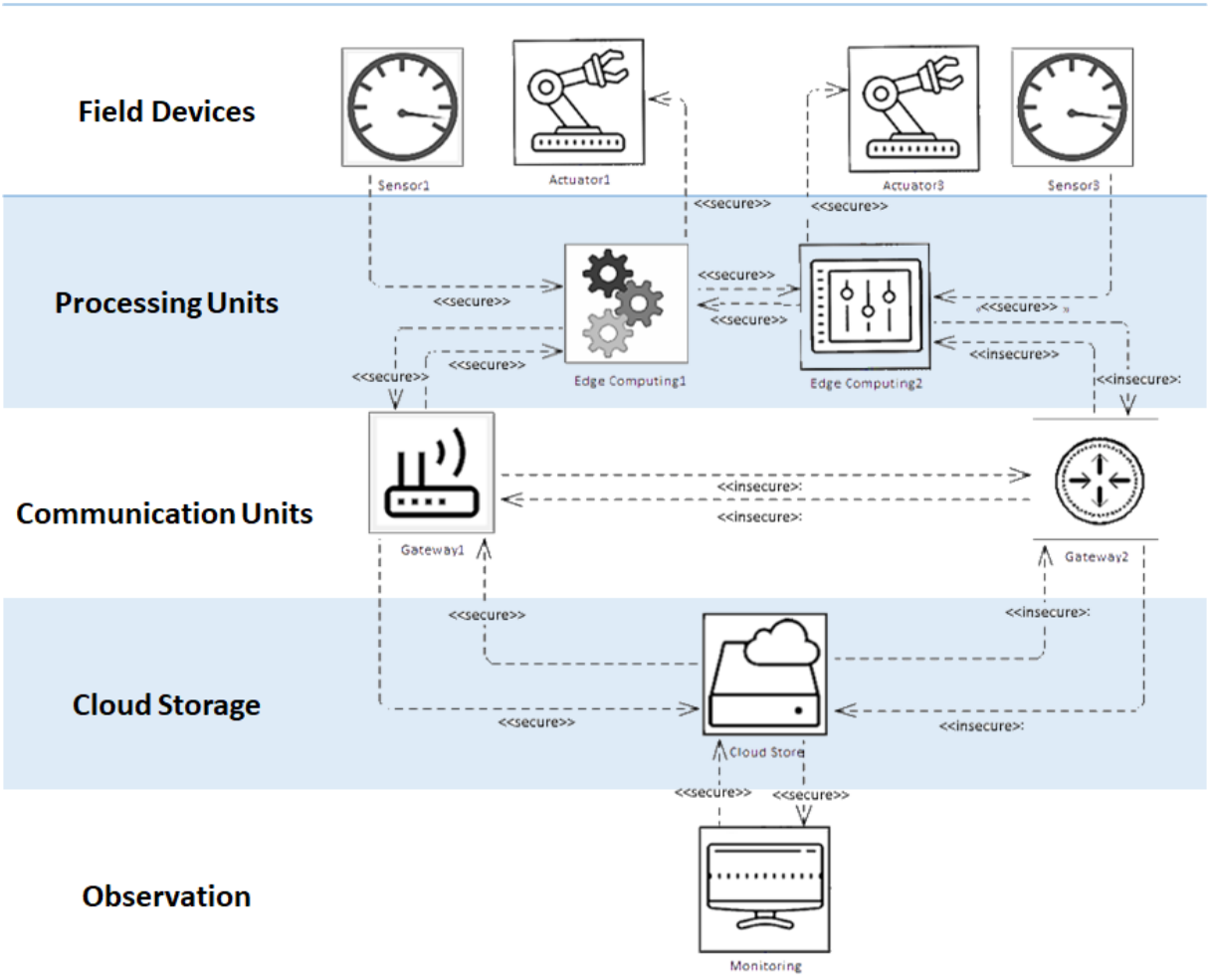


Figure 1: Smart factory example as a cyber-physical application

Figure 2 defines the main building blocks of the ontology-based security tool-chain (i.e., Threat Analysis, Security Requirements, Ontology Generator, and Security Testing).

2.1 Threat Analysis

Threat analysis is a method for analyzing and defining the potential threats which exploit the existing vulnerabilities in a system. The AIT Threat Management Tool (ThreatGet) is used in this phase to identify, describe, and understand several threats [5]. ThreatGet can be applied to a wide range of CPS applications such as smart factory, automotive, railways, networks, and others. The tool helps the system architect to:

- Build a secure CPS application,
- Identify security vulnerabilities,
- Identify threats and evaluate their risks.

In the analysis process, the tool uses the security configuration parameters of each unit in a system model. ThreatGet has a built-in threats catalog which contains a wide range of potential threats. The ThreatGet tool classifies the detected threats into six main groups according to the STRIDE model (i.e., Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service (DoS), and Elevation of Privilege) [6].

ThreatGet is applied to the "Smart Factory" example to define and detect potential threats. The tool identifies 71 potential threats by scanning all elements and connectors in the model. The existence of a legacy Gateway (Gateway2) and insecure communication protocols lead to generating 26 potential threats out of the total detected threats.

The Ontology Generator collects the threats data (i.e., category, severity, and source and target units) and converts these data into ontology entities such as classes, subclasses, individuals, and properties. That will be used in the security testing process.

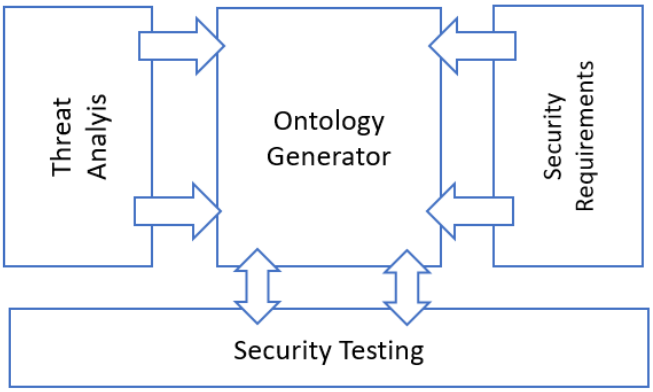


Figure 2: The building blocks of the ontology-based security tool-chain

2.2 Security Requirements

The security requirements play an essential role in any security engineering process. That aims to cover and handle the security weaknesses in a system to build a secure system according to a standardized model. The authors developed the Model-based Security Requirement Management Tool (MORETO) as a tool for security requirements analysis, allocation, and management using modeling languages such as SysML/UML. MORETO is an Enterprise Architect (EA) plugin for managing the IEC 62443 security standard [7] and IEEE 1686 security stand [8]. The primary purpose of the IEC 62443 series is to present a framework that addresses current and future security weaknesses in industrial systems and enables security risk management to the complete life cycle [4] [7]. IEEE 1686 describes the functions and characteristics to be implemented in Intelligent Electronic Devices (IEDs) to support Critical Infrastructure Protection (CIP) applications. The standard concerns the security issues according to the access, operation, configuration, firmware revision, and data retrieval from an IED [7]. The vulnerable element in this example is the Gateway2. MORETO selects 11 security requirements that must be applied to that example (Smart Factory) to handle the security weaknesses of the Gateway2.

The selected security requirements are presented in the ontological description to be used in the next phase to ensure that these requirements are fulfilled.

Security Testing

This phase uses the previously generated ontological representations of the identified potential threats and the selected security requirements to validate and verify the operational and the performance of the security requirements against the system security flaws. This phase creates an ontology linking between the hierarchical of threats and the security requirements nodes. This mapping process defines links between these two hierarchical ontologies, which represent that the selected security requirement can handle one or more of the detected threats.

Figure 3 shows the ontology representation of the detected threats on the "left-hand side," the selected security requirements on the "right-hand side," and the connections between these two hierarchies represent the security requirements are handled the security weaknesses (threats). That is considered as a comprehensive overview of all details of assets, detected threats, and related security requirements in the Smart Factory example. This comprehensive overview is called the Ontology Outlook. The security verification and validation process is achieved by using Ontology Security Testing Algorithm (OnSecta). The authors developed OnSecta as a rule engine performs logical rules from a set of asserted facts or axioms of threats and security requirements.

The algorithm performs security verification and validation in this example according to the current security status, and the actual security goal needs to be achieved. The Security Target (ST) is set during the concept phase to define a specific security goal. Therefore, the security requirements are used to mitigate the risk to an acceptable level. The resulting state is defined as Security Achieved (SA). The testing process completes if SA = ST; otherwise, OnSecta applies inference rules to the Ontology Outlook based on the value of SA and ST to finds additional security requirements that address threats. The security requirements are described in the ontology structure and stored in an Ontology Knowledge Base (KB).

First of all, the values of SA and ST have to be determined to define the current security state of the system (after applying the security requirements which are selected by MORETO) and to define the actual security target needs to be achieved. The value of ST can be determined by defining the maximum number of unacceptable risk severities in this example. We consider the threats with extreme severity are unacceptable, that means the ST = "the number of extreme threats." According to the identified potential threats, ThreatGet classifies 11 threads as extreme severity. Furthermore, the value of ST in this example equals 11.

OnSecta performs a series of queries to the Ontology Outlook to check which of the applied security requirements are fulfilled. According to the IEC 62443 security standard, the algorithm finds that the selected security requirements are addressed eight out of eleven extreme threats. The value of SA = 8 (SA <ST), and OnSecta should select additional security requirements until SA = ST.

2.3 Model Evaluation

The ontologies are considered the core of this work. That are used to represent data which are generated by ThreatGet and MORETO (threats and security requirements) to be used in the security testing process. OnSecta algorithm is applied to the Ontology Outlook to confirm that security requirements are meet the actual security target.

The following chart in Figure 4 describes the number of the selected security requirements of the Gateway2 (legacy device). MORETO and OnSects select the security requirements according to the IEC 62443-4-2, the technical security requirements for Industrial Automation and Control Systems Components (IACS) [4].

The IEC 62443 series provides detailed technical control system component requirements (CRs) associated with seven foundational requirements (FRs), there are a total of seven FRs [4] [9]:

- Identification and authentication control (IAC),
- Use control (UC),
- System integrity (SI),

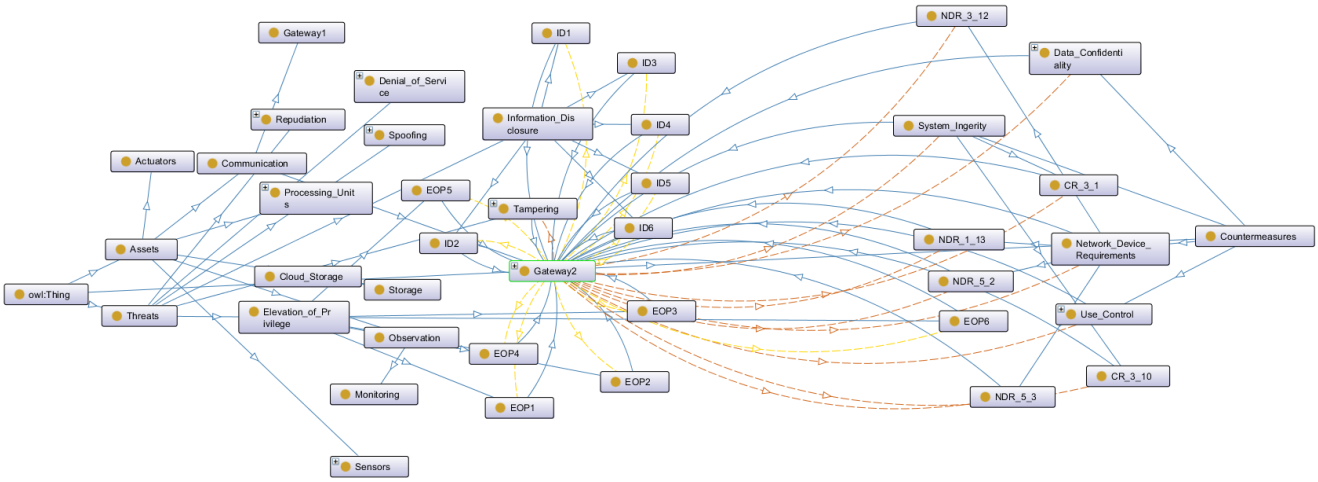


Figure 3: Ontology linking between threats (left) and security requirements (right)

- Data confidentiality (DC),
- Restricted data flow (RDF),
- Timely response to events (TRE),
- Resource availability (RA).

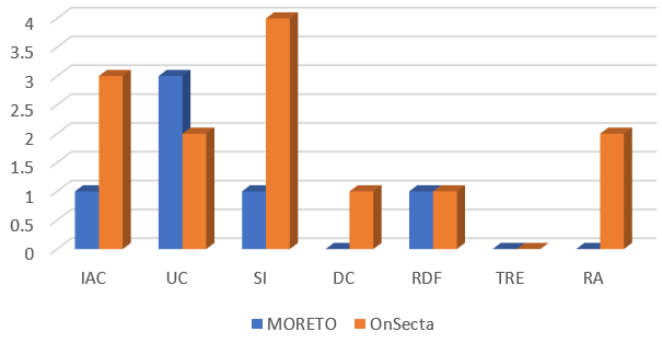


Figure 4: The number of the selected security requirements for the legacy Gateway2

The chart displays the number of security requirements according to the seven FRs which are selected to address the security weaknesses in the legacy device (Gateway2).

3 CONCLUSIONS

To conclude this contribution, the paper introduced a modern security tool-chain for critical CPS applications based on the ontology approach. The first phase in this chain is the ThreatGet. ThreatGet detects and identifies potential threats in critical systems. The second phase is MORETO tool as a security requirement management methodology to select security requirements to address the identified potential. The tool translates the data of threats and security requirements into ontological entities to be used in the security testing process. A smart factory example is used in this work to define the potential threats that can be generated from legacy components. The paper ends by identifying the number of the selected security requirements of legacy components (Gateway2) before and after applying the OnSecta.

OnSecta is still in the developing stage; the authors work on developing the different building blocks of OnSecta. Then, define a complete set of rules to perform security testing process.

ACKNOWLEDGMENT

This work has received funding from the AQUAS project, under grant agreement No. 737475. The project is co-funded by grants from Austria, Germany, Italy, France, Portugal and ECSEL JU.

REFERENCES

[1] Zhendong Ma, Aleksandar Hudic, Abdelkader Shaaban, and Sandor Plosz. Security viewpoint in a reference architecture model for cyber-physical production systems. In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 153–159. IEEE.

[2] Abdelkader Magdy Shaaban, Christoph Schmittner, Thomas Gruber, A. Baith Mohamed, Gerald Quirchmayr, and Erich Schikuta. CloudWoT - a reference model for knowledge-based IoT solutions. In *Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services - iiWAS2018*, pages 272–281. ACM Press.

[3] ISO/IEC. Information security management systems: Overview and vocabulary. International standard, International Organization for Standardization - ISO and International Electrotechnical Commission - IEC, Geneva-Switzerland, January 2014.

[4] IEC 62443-4-2. Industrial communication networks - network and system security -part 4-2: Technical security requirements for iaas components. Technical report, International Electrotechnical Commision, 2018.

[5] Austrian Institute of Technology. Threatget - threat analysis and risk management. https://www.threatget.com. Accessed: 29.06.2019.

[6] Adam Shostack. *Threat modeling: Designing for security.* John Wiley & Sons, 2014.

[7] Abdelkader Magdy Shaaban, Erwin Kristen, and Christoph Schmittner. Application of iec 62443 for iot components. In *International Conference on Computer Safety, Reliability, and Security*, pages 214–223. Springer, 2018.

[8] IEEE 1686. Ieee 1686-2013 - ieee standard for intelligent electronic devices cyber security capabilities. Technical report, Institute of Electrical and Electronics Engineers, 2013.

[9] ISA. Ansi/isa-62443-4-2-2018, security for industrial automation and control systems, part 4-2: Technical security requirements for iacs components, 2018. [accessed on: 2019.06.28].

White-Box and Black-Box Test Quality Metrics for Configurable Simulation Models

Urtzi Markiegi
umarkiegi@mondragon.edu
Mondragon Unibertsitatea
Mondragon, Spain

Aitor Arrieta
aarrieta@mondragon.edu
Mondragon Unibertsitatea
Mondragon, Spain

Leire Etxeberria
letxeberria@mondragon.edu
Mondragon Unibertsitatea
Mondragon, Spain

Goiuria Sagardui
gsagardui@mondragon.edu
Mondragon Unibertsitatea
Mondragon, Spain

ABSTRACT

Simulation models are widely employed to model and simulate complex systems from different domains, such as automotive. These systems are becoming highly configurable to support different users' demands. Testing all of them is impracticable, and thus, cost-effective techniques are mandatory. Costs are usually attributed either to the time it takes to test a configurable system or to its monetary value. Nevertheless, for the case of test effectiveness several quality metrics can be found in the literature. This paper aims at proposing both black-box and white-box test quality metrics for configurable simulation models relying on 150% variability modeling approaches.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software testing and debugging**; *Software verification and validation.*

KEYWORDS

Product Lines, Test Quality Metrics, Simulation Models

ACM Reference Format:
Urtzi Markiegi, Aitor Arrieta, Leire Etxeberria, and Goiuria Sagardui. 2019. White-Box and Black-Box Test Quality Metrics for Configurable Simulation Models. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3307630.3342396

1 INTRODUCTION

Currently, the development of many complex systems, such as Cyber-Physical Systems (CPSs) or automotive systems, is driven by employing simulation models. The use of these simulation models is even more important when such systems are highly configurable. This is mainly because it is unfeasible to develop initial prototypes during initial verification and validation processes due to the fact that such systems can be configured into thousands to millions configurations. Thoroughly testing these systems is also unfeasible because apart from the vast amount of configurations to be tested, running a simulation is also expensive. Notice that running a test case in such simulations, unlike unit testing, takes from seconds to minutes. This is because, firstly, the simulation is performed at the system level and secondly, the simulation models typically encapsulate complex mathematical models that simulate continuous behavior of the physical part of the system [5]. Even in some cases, co-simulation is required to integrate simulation models of several Original Equipment Manufacturers (OEMs), or to simulate complex systems. For instance, in our previous paper [16], we show how we distributed the simulation of several elevators in different computers by using co-simulation. While co-simulation enables high flexibility to engineers, the interaction and communication of several simulators increases the overall simulation time.

In the context of MATLAB/Simulink models, which is the driving simulation tool in the context of CPSs, several papers have proposed different variability modeling alternatives [2, 6, 8, 15]. Also, for this tool, different techniques have been proposed to efficiently generate test cases, select or prioritize them [4, 13]. The test generation, selection and prioritization of these test cases is guided by measuring the quality of test cases (e.g., by measuring white-box coverage). However, in the context of product line engineering, to the best of our knowledge, there have not been proposals that define test quality metrics for simulation models. In this paper, we propose a set of test quality metrics for simulation models that rely on both, white-box and black-box metrics.

The paper is structured as follows. In Section 2 we introduce background of simulation models and formalize the notations we will use thorough the paper. In Section 3 we propose a set of white-box and black-box quality metrics that aim at measuring the quality of tests in configurable simulation models. In section 4 we give an overview of potential test applications for the proposed quality metrics. Lastly, Section 5 concludes the paper.

2 FORMALIZATION

Developers of complex systems (e.g., CPSs) usually rely on simulation to create a digital model that predicts the performance of their designs in the real world. In this context, CPS simulation tools follow the Model-Based Design (MBD) work-flow and are typically

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342396

data-flow models, each model containing a set of blocks. These blocks accept data through their inputs, and after performing a set of operations (e.g., mathematical, logical, etc.) produce some outcomes which are passed through their outputs. Hierarchies are specified in these models through different levels by using an approach that relies on subsystems. When modeling variability of simulation models, different approaches can be followed (e.g., delta modeling approaches [8] or negative variability modeling based on 150% models [3]). This paper is limited to 150% models and thus, adaptions will be required for other variability approaches.

We refer to 150% models to a domain engineering asset that represents the whole product line. 150% models integrate all the variability, i.e., the variability related to the entire configurable model into one single model [3]. In addition, the features of the product line are associated with specific blocks of the 150% model. When a specific product variant is selected, the variability of the 150% model is bound, forming the 100% model (i.e., the model specific to that configuration) [3]. Hence, a specific product configuration will have associated features of the blocks it contains.

Let $SM_{150\%} = (sm_1, sm_2, ..., sm_N)$ be a 150% simulation model that can be configured in N simulation models. For black-box test quality metrics we refer to those metrics that can be derived by considering the inputs and outputs of the simulation model. $SM_{150\%}$ is composed of input and output signals. $I_{150\%} = \{i_1, i_2, ..., i_{N_{in}}\}$ is a subset of inputs signals and $O_{150\%} = \{o_1, o_2, ..., o_{N_{out}}\}$ is a subset of output signals for the 150% model. Notice that since $SM_{150\%}$ represents the entire product line, N_{in} and N_{out} represent all the inputs and outputs of the configurable simulation model. White-box test quality metrics rely on internal coverage criteria (e.g., branch coverage, which measures the number of branches exercised for a given set of test cases). These coverage criteria define some precise test objectives to be covered [10]. In simulation models, the test objectives are located in blocks. For example, in a switch type block we have two test objectives, one test objective for the case in which it is true and another for the case in which it is false. Let $TO_{150\%} = \{to_1, to_2, ..., to_{N_{to}}\}$ be a set of N_{to} test objectives that represent the test objectives of the entire product line.

Let $I_{sm_i} = \{i_1, i_2..., i_{N_{in_i}}\}$ be a set of N_{in_i} inputs of the configuration model sm_i, where i_a can be any signal from $I_{150\%}$ (i.e., $i_a \in I_{150\%}$) and $N_{in_i} \leq N_{in}$. Accordingly, let $O_{sm_i} = \{o_1, o_2..., o_{N_{out_i}}\}$ be a set of N_{out_i} outputs of the configuration model sm_i, where o_a can be any signal from $O_{150\%}$ (i.e., $o_a \in O_{150\%}$) and $N_{out_i} \leq N_{out}$. For white-box coverage, test objectives are measured, holding similarly to inputs and outputs: let $TO_{sm_i} = \{to_1, to_2..., to_{N_{to_i}}\}$ be a set of N_{to_i} test objectives of the configuration model sm_i, where to_a can be any test objective from $TO_{150\%}$ (i.e., $to_a \in TO_{150\%}$) and $N_{to_i} \leq N_{to}$.

3 TEST QUALITY METRICS FOR CONFIGURABLE SIMULATION MODELS

In this section we define the test quality metrics that can assess the effectiveness of a test case or a set of test cases when testing configurable simulation models. We divide the section in white-box and black-box test quality metrics, and adapt the equations to the context of configurable simulation models.

3.1 White-box test quality metrics

The number of test objectives in a system can depend on the type of coverage it is aimed at to be employed (e.g., data-flow or control-flow) as well as the metrics (e.g., Condition Coverage or Decision Coverage).

Table 1: Table showing relation between test objectives, features and test cases.

	f1		f2			f3		f4
	to1	to2	to3	to4	to5	to6	to7	to8
TC1	X		X		X	X		
TC2		X		X		X	X	
TC3	X	X			X			X
TC4	X	X	X			X		X
TC5			X	X				X
TC6	X		X				X	
TC8		X		X				
TC9				X	X		X	X

3.1.1 Structural coverage. The testing community has widely employed structural coverage as a quality measure [12, 18]. The purpose of the structural coverage is to determine the amount of code (or model) that has been exercised during a testing activity. In our context, the coverage is measured in terms of covered test objectives that are associated with product line assets. When specific product configurations are derived from the product line, a particular number of test objectives are implicitly derived. The number of test objectives in the specific product is equal to or less than the number of test objectives of the entire product line. The structural coverage can be measured in each configuration at the application engineering level as well as at the domain engineering level. Refer to [11] to have a detailed description of how to measure the structural coverage at the application and domain engineering level.

Consider Table 1 as an example, which presents the relation among 4 features (i.e., f_1, f_2, f_3, f_4) along with eight different test objectives (to) with nine different test cases (i.e., TC1 to TC9). Let us suppose that product P_k is obtained after deriving the product line configured with two features $P_k = \{f_1, f_2\}$. Therefore, product P_k contains five test objectives $TO_{P_k} = \{to_1, to_2, to_3, to_4, to_5\}$ associated to product assets. Let us suppose also a test suite composed by the test cases TC1 and TC2 that exercise the P_k product $TS_{P_k} = \{TC1, TC2\}$. The structural coverage obtained by the product at the application engineering level (i.e., at the product level) will consist of the number of test objectives covered in relation to the number of test objectives in the product. In this case, the five test objectives of the product will be covered (i.e., 100% structural coverage of the product will be achieved). However, the structural coverage obtained by the product with the same test suite at the domain engineering level will consist of the number of test objectives covered relative to the total number of objectives of the entire product line. In this case, the number of covered test objectives is the same (i.e., 5 test objectives) but there are 0 test objectives in

the entire product line, so the achieved structural coverage at the application engineering level would be 5/8 = 62%.

3.1.2 Feature coverage. Test objectives are associated to product line assets, and these assets are associated to features. Bearing this in mind, it is possible to obtain the extent to which a feature has been tested in a simulation model. Let us suppose that $TO_{f_i} = \{to_1, to_2, ..., to_{N_{f_i}}\}$ are the test objectives covering feature i, and these test objectives are part of the entire product line, and thus can appear in any configuration (i.e., $to_a \in TO_{150\%}$). Intuitively, the coverage for a given feature is the percentage of its test objectives that have been covered (i.e., $\sum_{i=0}^{N} to_i/N_{f_i}$).

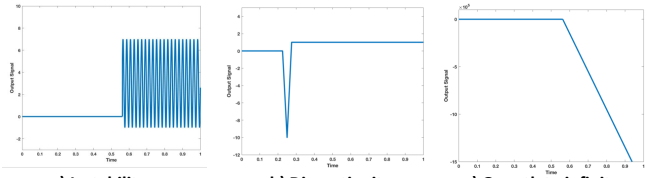
Considering again the example in Table 1, the first test case (i.e., $TC1$) would obtain a test coverage for the first feature of 50%, as it only covers $to1$, 66.66% for the second feature as it covers two test objectives out of three, 50% for the third feature and 0% of feature coverage for the fourth feature. If we complement the first test case with the second, we have a 100% of feature coverage for $f1$, $f2$ and $f3$, since $TC2$ covers those test objectives not covered by $TC1$ for these first three features. However, the feature coverage for $f4$ remains at 0%. To obtain a full feature coverage, we need to complement these two test cases with one of the test cases covering $to8$, which is the test objective related to $f4$; thus, either $TC3$, $TC4$, $TC5$ or $TC9$ can be selected. The feature coverage of the entire product line will be the sum representation of the feature coverage obtained by each feature (i.e., $\sum_{i=0}^{N} cov(f_i)/Nf$, being $cov(f_i)$ the coverage obtained for feature i and Nf the total number of features in the product line).

3.1.3 Feature pairwise coverage. It is well known that many faults in product line engineering appear due to the interaction of pairs of features. In a simulation model, this could be measured by considering the interaction of test objectives when these are associated to different features. When considering Table 1, 6 feature pair interactions exist (i.e., f1-f2, f1-f3, f1-f4, f2-f3, f2-f4, f3-f4). If we consider the coverage for the first interaction (i.e., interaction of f1-f2), the interaction between the test objectives of these features would need to be considered (i.e., to1-to3, to1-to4, to1-to5, to2-to3, to2-to4 and to2-to5). Notice that in this case, selecting TC1, two of the objective interactions have been covered, and thus, a 33.33% of feature pairwise coverage is scored for the interaction of f1 and f2. Similar to the feature coverage, the feature pairwise coverage of the entire product line is calculated as the sum representation of the coverage obtained by each feature pair (i.e., $\sum_{i=0}^{N}(\sum_{j=i+1}^{N} fpcov(f_i, f_j))/Nfpair$, being $fpcov(f_i, f_j)$ the feature pair coverage for features i and j and $Nfpair$ the total number of feature pairs in the product line).

3.2 Black-box test quality metrics

In [13, 14], different black-box metrics are proposed. These metrics are based, on the one hand on anti-patterns of simulations models, and on the other hand on similarity metrics based on the euclidean distances of inputs and outputs. In our previous paper [4], we adapted these metrics for the test case selection context in the field of simulation models. In this paper, we aim at proposing these metrics for the context of configurable simulation models.

Figure 1: Anti-patterns for simulation models



3.2.1 Test quality metrics based on anti-patterns. Three different anti-patterns were presented by Matinnejad et al. in [14], which can be seen in Figure 1. Instability aims at measuring quick and frequent oscillations of a signal [14]. Discontinuity is an anti-pattern in which an output signal shows a short duration pulse [14]. Lastly, growth to infinity is an anti-pattern where an output signal shows how a signal grows to an infinite value [14].

In our previous paper [4], we adapted those anti-patterns proposed in [13, 14] to be normalized independently of the number of outputs a model had and the data-type and maximum/minimum values of their outputs. Refer to [4] for further detail of how each anti-pattern is measured for each signal.

For configurable simulation models, the anti-pattern degree for each of the test cases given a test suite (i.e., TS) is measured by observing each output of the simulation model. However, since each output in O_{150} can represent a particular variable in different units (e.g., one output can represent vehicle speed in km/h while another output can represent the acceleration in m/s^2), it is mandatory to normalize the anti-pattern degree of each test case.

Given a 150% simulation model with N_{out} outputs (i.e., $O = \{o_1, o_2, ..., o_{N_{out}}\}$), the anti-pattern degree of a test case j in TS is obtained with Equation 1, where $ap(Osig_{j_i})$ is the anti-pattern degree of the i-th signal for tc_j and $\max(ap(Osig_i))$ is the maximum anti-pattern degree obtained in the i-th signal when considering all test cases in TS. Notice that ap can be either instability, discontinuity or growth to infinity.

$$TCap(tc_j) = \frac{1}{\max(ap(Osig_i))) \cdot M} \times \sum_{i=1}^{M}(ap(Osig_{j_i}) \quad (1)$$

3.2.2 Similarity. It is well known that a set of different test cases is more likely to detect faults [7, 9]. By following this principle we have defined two similarity measures for configurable systems. Firstly, we have defined the similarity between test cases adapting the Euclidean distance measure proposed for simulation models [4, 13, 14]. In our previous paper [4] the details of how to measure two test cases with the Euclidean distance is detailed.

Equation 2 defines the input signal-based distance between two different test cases (TC_a and TC_b) for a simulation model with N inputs (in the 150% simulation model). When comparing two test cases executed in two product configurations, three group of signals must be considered in simulation models: (i) the signals that both configurations share, (ii) the signals that are present only in one configuration (but not in the other) and (iii) the signals that both configurations do not have. Equation 2 details how to measure the distance between two test cases (i.e., TC_a and TC_b). $D(sig_{a_i}, sig_{b_i})$ is referred to the Euclidean distance between the

signal i of test case a and test case b, where S is the number of signals that both configuration models share (i.e., the first group). We consider that a distance between two signals is the maximum when a configuration has one specific signal and the other one does not have it. The maximum distance (maxD) will be 1, and d will be the number of signals that a specific configuration has but the other configuration doesn't. Lastly, if both configurations do not have a specific signal, we consider that from the point of view of the test case similarity, both are the same, and we consider the minimum distance (which is 0 and thus, not considered in the equation).

$$TCD(TC_a, TC_b) = \frac{\sum_{i=1}^{s} D(sig_{a_i}, sig_{b_i}) + \sum_{i=1}^{d} maxD}{N} \quad (2)$$

The previous equation can also be complemented with the similarity that two independent configurations have in terms of features (e.g., by considering the similarity measure proposed by Al-Hajjaji [1]). For instance, given a configuration u (C_u) and a configuration v (C_v), and two test cases (e.g., TC_a and TC_b), where TC_a is tested on C_u and TC_b on C_v, the distance between both test cases and configurations can be measured as a weighted sum of both. Where the distance between both test cases (i.e., $TCD(TC_a, TC_b)$) is calculated following for instance Equation 2 and the distance between both products (i.e., $PD(P_u, P_v)$ as given in Equation 3. Notice that Wp is the weight given to the distance between both products whereas Wtc is the weight given to the distance between test cases.

$$Sim(P_u, TC_a, P_v, TC_b) = PD(P_u, P_v) \cdot Wp + TCD(TC_a, TC_b) \cdot Wtc \quad (3)$$

4 POTENTIAL APPLICATIONS

Testing configurable simulation models is expensive [3, 5]. As a result, verification and validation activities must consider several cost-effectiveness measures at different levels. The proposed quality metrics can be used as effectiveness measures for various testing activities combined with cost measures. These activities can include, for instance, test generation and regression test selection and prioritization activities.

As for test generation, these metrics can be used as heuristics along with already consolidated search algorithms. It would also be necessary to consider cost measures, which are typically associated to the test execution time [5]. As for test selection and prioritization, in prior works test history has been considered [5, 17]. Notice, however, that history-based effectiveness measures require a long start-up time to be effective. On the contrary, the metrics proposed in this paper do not require long historical execution. However, it should also be considered that some of the proposed measures (e.g., white-box metrics) may have scalability problems.

5 CONCLUSION AND FUTURE WORK

In this paper we propose several quality metrics for the context of configurable simulation models. These quality metrics can be used to determine how thoroughly a configurable simulation model has been tested. In addition, they can also be used to guide the search in different testing activities, including test generation, test selection and test prioritization. Unlike history-based test quality techniques, black-box and white-box techniques have the advantage of not requiring a large start-up to be effective. This advantage increases the possibility of using these metrics at early verification and validation stages.

In the future, we would like to investigate regression test selection and prioritization techniques with all these different techniques to select and prioritize test cases. To this end, different benchmarks and models will need to be investigated.

REFERENCES

[1] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. Effective product-line testing using similarity-based product prioritization. *Software & Systems Modeling*, 18(1):499–521, 2019.

[2] Manar H Alalfi, Eric J Rapos, Andrew Stevenson, Matthew Stephan, Thomas R Dean, and James R Cordy. Semi-automatic identification and representation of subsystem variability in simulink models. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 486–490. IEEE, 2014.

[3] Aitor Arrieta, Goiuria Sagardui, Leire Etxeberria, and Justyna Zander. Automatic generation of test system instances for configurable cyber-physical systems. *Software Quality Journal*, 25(3):1041–1083, 2017.

[4] Aitor Arrieta, Shuai Wang, Ainhoa Arruabarrena, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. Multi-objective black-box test case selection for cost-effectively testing simulation models. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1411–1418. ACM, 2018.

[5] 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.

[6] Danilo Beuche and Jens Weiland. Managing flexibility: Modeling binding-times in simulink. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 289–300. Springer, 2009.

[7] R. Feldt, S. Poulding, D. Clark, and S. Yoo. Test set diameter: Quantifying the diversity of sets of test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 223–233, April 2016.

[8] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-class variability modeling in matlab/simulink. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, page 4. ACM, 2013.

[9] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Achieving scalable model-based testing through test case diversity. *ACM Trans. Softw. Eng. Methodol.*, 22(1):6:1–6:42, March 2013.

[10] Michaël Marcozzi, Sébastien Bardin, Nikolai Kosmatov, Mike Papadakis, Virgile Prevosto, and Loïc Correnson. Time to clean your test objectives. In *Proceedings of the 40th International Conference on Software Engineering*, pages 456–467, 2018.

[11] Urtzi Markiegi, Aitor Arrieta, Leire Etxeberria, and Goiuria Sagardui. Test case selection using structural coverage in software product lines for time-budget constrained scenarios. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 2362–2371, New York, NY, USA, 2019. ACM.

[12] Martina Marré and Antonia Bertolino. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering*, 29(11):974–984, 2003.

[13] Reza Matinnejad, Shiva Nejati, Lionel Briand, and Thomas Bruckmann. Test generation and test prioritization for simulink models with dynamic behavior. *IEEE Transactions on Software Engineering*, 2018.

[14] Reza Matinnejad, Shiva Nejati, and Lionel C Briand. Automated testing of hybrid simulink/stateflow controllers: industrial case studies. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 938–943, 2017.

[15] Thorsten Pawletta, Artur Schmidt, Bernard P Zeigler, and Umut Durak. Extended variability modeling using system entity structure ontology within matlab/simulink. In *Proceedings of the 49th Annual Simulation Symposium*, page 22. Society for Computer Simulation International, 2016.

[16] Goiuria Sagardui, Joseba Agirre, Urtzi Markiegi, Aitor Arrieta, Carlos Fernando Nicolás, and Jose María Martín. Multiplex: A co-simulation architecture for elevators validation. In *2017 IEEE International Workshop of Electronics, Control, Measurement, Signals and their application to Mechatronics*, pages 1–6, 2017.

[17] Shuai Wang, Shaukat Ali, and Arnaud Gotlieb. Minimizing test suites in software product lines using weight-based genetic algorithms. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1493–1500. ACM, 2013.

[18] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

Enabling Efficient Automated Configuration Generation and Management

Sebastian Krieter
University of Magdeburg, Harz University of Applied Sciences
Magdeburg, Wernigerode, Germany
sebastian.krieter@ovgu.de

ABSTRACT

Creating and managing valid configurations is one of the main tasks in software product line engineering. Due to the often complex constraints from a feature model, some kind of automated configuration generation is required to facilitate the configuration process for users and developers. For instance, decision propagation can be applied to support users in configuring a product from a software product line (SPL) with less manual effort and error potential, leading to a semi-automatic configuration process. Furthermore, fully-automatic configuration processes, such as random sampling or t-wise interaction sampling can be employed to test or to optimize an SPL. However, current techniques for automated configuration generation still do not scale well to SPLs with large and complex feature models. Within our thesis, we identify current challenges regarding the efficiency and effectiveness of the semi- and fully-automatic configuration process and aim to address these challenges by introducing novel techniques and improving current ones. Our preliminary results show already show promising progress for both, the semi- and fully-automatic configuration process.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

Configurable System, Software Product Lines, T-Wise Sampling, Uniform Random Sampling, Decision Propagation

ACM Reference Format:
Sebastian Krieter. 2019. Enabling Efficient Automated Configuration Generation and Management . In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3307630.3342705

1 INTRODUCTION

Todays industrial software product lines comprise thousands of features [6, 7], which makes crucial tasks, such as automated analysis and configuration management, challenging [4]. Configuration management is one of the fundamental tasks for developers of software product lines. In order to generate a product from a set of implementation artifacts a configuration needs to define a valid selection of features [2, 9]. Nevertheless, creating configurations is not only done by end users, who want to customize their software product, but is used by developers throughout the entire process of software product line engineering (SPLE). Besides generating a software product, configurations are, for instance, also used for variability analysis, process testing, and parameter optimization. At the same time, creating a valid configuration can be a challenging task due to complex constraints within a feature model.

For our research, we categorize the configuration process into three kinds of automation, namely *manual*, *semi-automatic*, and *fully-automatic*. In the manual configuration process users select all desired features by their selves, and thus, no decision is made by an algorithm. This gives users total control over a configuration, but also requires them to fully comprehend all constraints from the feature model and separately check whether their final configuration is valid. Especially for large feature model, this makes this process very tedious and error-prone [4].

To facilitate the manual configuration process users can employ several techniques to automatically select or deselect certain features, which leads to a semi-automatic configuration process. There are many different techniques that can support users in creating a configuration. For instance, an algorithm can recommend to select or deselect certain features [30], select or deselect implied features (i.e., decision propagation) [19], auto-complete a partial configuration, or repair errors that cause a configuration to be invalid [14]. These techniques mitigate the risk of creating invalid configurations and decrease the manual effort for users, while still providing them sufficient control over their created configurations.

In a fully-automatic process, configurations are generated entirely by an algorithm based on certain input specifications. For instance, an algorithm can generate random configurations (e.g., for testing purposes), configurations that achieve a satisfy a certain coverage criterion (e.g., t-wise interaction coverage), or configuration that are similar to previously created configurations (e.g., for parameter optimization) [28]. A fully-automatic process requires no manual effort for creating a configuration. On the other hand, users only have limited control over the created configurations, as they entirely depend on the chosen algorithm and input data. Thus, a fully-automatic process often serves other tasks within SPL development than creating a single user configuration (e.g., product testing, parameter optimization, or evaluation).

Having tool support to generate configuration is desirable as it substantially decreases manual effort. However, many algorithms and techniques that are used for this purpose still do not scale well to

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342705

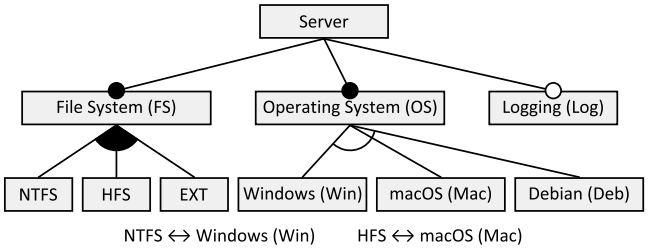


Figure 1: Feature model for *Server* SPL.

large feature models. Thus, in our research, we want to identify and address current challenges for the fully- and semi-automated configuration process. In particular, we consider decision propagation, as it is one of the most helpful techniques for the semi-automatic configuration process and can also be used in automated analyses and algorithms for the fully-automatic configuration process. Furthermore, we consider random sampling and t-wise interaction sampling as means for the fully-automatic configuration process. Finally, we want to consider the aforementioned techniques with regard to feature model evolution.

2 FUNDAMENTALS AND MOTIVATION

Before we describe our research plan, we provide a brief description of our notion of features models and configurations. Furthermore, we explain the semi- and fully-automatic configuration process in more detail. To this end, we focus on the techniques that we want to investigate and improve, such as decision propagation, uniform sampling, and t-wise sampling. We describe their usefulness and list current challenges. In particular, we look at the efficiency and effectiveness of these techniques.

2.1 Feature Models

A feature model represents the variability of a software product line by defining the set of features and their interdependencies [2, 3, 9]. In our work, we only consider boolean features, which can be either select or deselected within a configuration. A common representation of feature models are feature diagrams [10], as for example depicted in Figure 1. Each node in a feature digram represent a feature and each edge a relationship between two or more features. In addition, there can also be cross-tree constraint, that cannot be included within the tree hierarchy.

Feature models can also be expressed by propositional formulas [3, 10]. This is useful, as most analyses regarding the feature model (e.g., whether it is valid) can be reduced to the boolean satisfiability problem (SAT) and then be solved by using a dedicated satisfiability solver. Usually, these solvers operate on propositional formulas in conjunctive normal form (CNF) [25]. Every feature model that uses boolean constraints can be converted into a CNF [3, 10, 18].

2.2 Configuration

A configuration represents the set of selected and the set of deselected features from a given feature model [2]. It is used to generate a product from an SPL.

For our research, we consider two classifications for configuration, *validity* and *completeness*. A configuration must not contain every feature of a feature model. If there are features from the feature model, for which a configuration does not define whether they are selected or deselected, the configuration is called *partial*. Otherwise, if a configuration defines an assignment for each feature from the feature model, it is called *complete*. A configuration is called *valid*, if its defined selection of features fulfills all constraints from the feature model. Otherwise, it is called *invalid*. The set of all valid configurations of a feature model forms the problem space of an SPL, while the set of products forms the solution space.

Analogous to feature models, many analyses on configurations can be reduced to SAT (e.g., whether a partial configuration contains a contradiction) [16].

2.3 Semi-Automatic Configuration Process

In the semi-automatic configuration process users are supported by certain techniques to minimize their manual effort. Within our research, we consider the technique of *decision propagation*, which helps users to create valid configuration more efficiently and can also be applied in the fully-automatic configuration process.

Decision Propagation. Decision propagation determines the selection and deselection of features based on a partial configuration [15, 24]. The most common use case for decision propagation is the application during the configuration process of a user. After a user selects or deselects a feature (i.e., makes a decision), decision propagation automatically calculates the implications on the remaining undefined features and complements the partial configuration accordingly. Applying decision propagation throughout the configuration process has several advantages. First, given a valid feature model, the user cannot configure a contradiction within the configuration. Thus, a complete configuration will always be valid, if decision propagation is applied after each decision. Second, it reduces manual effort, as users do not have to manually select or deselect features that are implied by their other selections. Third, it makes the consequences of a decision immediately visible to the users. Thus, if users are not satisfied with the resulting feature selection, they can immediately undo their manual decisions and select different features.

Besides creating a configuration from scratch, there are also other uses cases for decision propagation. It can be used in analyses that investigate the relationship between two ore more features (e.g., determining false-optional features [5]). It can also be used in sampling algorithms to update a partial configuration during the sampling process [1].

However, current implementations for decision propagation do not scale for large feature models. For instance, a straight-forward approach to implement decision propagation is to use a satisfiability solver to determine for each undefined feature in a partial configuration individually, whether it needs to be selected or deselected. This may take a relatively long time to compute as for large feature models, the number of calls to the satisfiability solver increases (due to potentially more undefined features) and, in addition, the satisfiability solver requires more time to compute a solution. In a real-time configuration process, waiting for several seconds or

even minutes after a decision is simply not feasible. This leads to our first challenge.

Challenge 1. *How can we enable efficient decision propagation for large feature models?*

In our research, we want to address this challenge by improving current algorithms for decision propagation. We intend to reduce computation time by reducing the number of necessary calls to the satisfiability solver. For this particular challenge, we already achieved promising results (cf. Section 5.1).

2.4 Fully-Automatic Configuration Process

In the fully-automatic configuration process configuration are generated without any manual effort from a user. There are several techniques to generate a single configuration or a list of configurations based on a certain user input. Within our research, we consider *uniform random sampling*, as it is a fundamental technique to generate configurations without further domain knowledge, and *t-wise interaction sampling*, as it is a well-established technique for computing a representative configuration sample of an SPL.

Uniform Random Sampling. Random sampling generates a list of complete and valid configurations that are randomly chosen from the problem space [34]. Users only have to provide a feature model and a desired sample size (i.e., number of configurations).

There are many different applications for random sampling regarding SPLE [34]. A common application is product-based testing. Generating a product from a random configuration and testing it can be an effective way to find variability bugs within an SPL without the need for advanced sampling techniques or further domain knowledge. Similarly, random sampling can also be used for optimizing non-functional properties of a product by trying different feature selections. For instance, it can be used by optimization algorithms to provide one or more starting points or act as a mutation operator to adapt existing configurations [12, 13]. Further, random sampling can be employed in the evaluation of novel algorithms, for instance for automated analyses of SPLs. Random samples can act as a baseline to serve as a control group or provide a benchmark. Another application for random sampling is auto completion of partial configurations. If a partial configuration must be completed and the remaining feature selections do not matter for the current task at hand, a random assignment is a suitable option.

Most of the aforementioned applications benefit from a random sample that is uniformly distributed. That means that it is equally likely for any valid configuration to occur in a given random sample. If the sample is not uniformly distributed, it may lead to undesirable results. For instance, if a biased random sample is used as input for testing, it is hard to detect bugs that only occur in unlikely configuration.

However, creating a uniform random sampling is a challenging task due to the constraints of a feature model, which limit the problem space. Therefore, a random configuration cannot simply be created by randomly choosing a subset of selected features, as the resulting configuration is likely to be invalid [21]. As a result, current algorithms for random sampling are either not uniform or require an enormous amount of memory or computation time [31]. For instance, a straight-forward algorithm for random sampling is to enumerate all valid configurations and randomly choose a subset from the resulting list. This is unfeasible for larger feature models, as the number of configurations normally grows exponentially with regard to the number of features, which makes it practically impossible to list all configurations for a large feature model [11]. On the other hand, a fast method for randomly creating a configuration is to use a satisfiability solver, which can provide a satisfying solution (i.e., a configuration). When the satisfiability solver is adapted in such a way that it uses non-deterministic feature selections during its search for a solution, the result is a random configuration. However, this method is dependent on the internal architecture of the employed solver and likely to biased towards certain configurations. This is due to the fact that some features are more likely to appear in a valid configuration than others. These circumstances lead to our next challenge.

Challenge 2. *How can we achieve efficient uniform random sampling?*

T-Wise Interaction Sampling. T-wise interaction sampling generates a set of configurations in such a way that all possible interactions of t features are contained within at least one configuration in the set [34]. For instance, considering the two features Win and Deb from Figure 1, there are three possible two-wise (i.e., pair-wise) interactions according to the feature model: Win and Deb are deselected, only Win is selected, and only Deb is selected. It is not possible to select both features simultaneously due to their mutually exclusive relationship. In pair-wise sampling each of these three interactions would be contained in at least one configuration.

The applications for t-wise interaction sampling are similar to random sampling. Mainly, it can be used for product-based testing. In fact the idea of t-wise interaction sampling is to include every possible feature interaction up to a certain degree in at least one product, which can then be tested. T-wise interaction sampling can also be used for optimizing non-functional properties for a product, as it enables estimating the impact of different feature interactions on the non-functional properties.

An issue with t-wise interaction sampling is to find a suitable trade-off between the degree of covered interactions (i.e., the value for t) and the resulting sample size. Naturally, as the value for t increases, the sample size increases as well. While a higher value for t (e.g., $t > 2$) may lead to more effective samples (e.g., samples that reveal more faults), a large configuration sample may be unfeasible to test [13]. This leads to our next challenge.

Challenge 3. *How can we compute effective and efficient samples for t-wise interaction sampling?*

2.5 Evolution of Feature Models

As with regular software, SPLs change over time, due to code maintenance and updates. Naturally not only the implementation artifacts of an SPL evolve, but also its feature model by adding and removing features and modifying their dependencies [8, 22, 27]. Thus, existing user configurations and configuration samples can become invalid and have to be updated accordingly. Ideally, the intentions of the original configurations are preserved as much as possible.

Regarding automatically generated configurations, they could be recomputed by using the same algorithm with the same input. However, this might be too expensive in terms of computation time. A better approach would be to take the actual changes made to the feature model into account to only update the existing configurations. This leads to our final challenge.

Challenge 4. *How can we can we efficiently update existing configurations and analysis results when a feature model evolves?*

3 PROBLEM STATEMENT AND RESEARCH QUESTIONS

Real-world feature models tend to be large and complex and are even growing (i.e., evolving) over time [18, 26]. Most of our outlined challenges regard scaling existing approaches for automated configuration generation to these real-world models. As we described above a general issue with most algorithms is their scalability. For large feature models they either require too much computation time, too much memory space, or simply do not compute feasible results. We can formulate our research questions in terms of efficiency and effectiveness of the techniques. However, both, effectiveness and efficiency are dependent on the task at hand.

For Challenge 1, we are concerned about the computation time to apply decision propagation to a given partial configuration. Thus, we formulate our first research question:

RQ_1 *How can the computation time of decisions propagation be decreased?*

Regarding Challenge 2, we consider the efficiency of uniform random sampling as the time it takes to compute a single random configuration or configuration sample of a fixed sized. As effectiveness, we consider the actual distribution of the randomly generated configurations. The closer the actual distribution is to a uniform distribution, the more effective the sampling algorithm. Thus, our second research question is as follows:

RQ_2 *How can the computation time of random sampling be decreased while maintaining a reasonably well uniform distribution?*

For Challenge 3, we can differentiate between two types of efficiency: the computation time required to compute a t-wise sample and the resulting sample size. Naturally, we want to decrease both values for a higher efficiency of the sampling algorithm. The effectiveness depends on the application of the t-wise sample. For instance, if employed for product-based testing, the effectiveness of a sample would be the potential number of faults it is able to reveal. The more faults can possibility be detected, the more effective is the sample and, in turn, the sampling algorithm. Hence, we formulate our third research question:

RQ_3 *Can the sample size and computation time of t-wise samples be decreased while maintaining or improving its effectiveness?*

In Challenge 4, we consider the computation time required to adapt previously generated configuration samples after a feature model evolution. The original purpose of these configurations, whether it be testing, optimization, or something different, should be kept. Thus, our research question is as follows:

RQ_4 *Can existing configuration samples be updated faster when considering the changes to the feature model?*

4 RESEARCH METHODOLOGY AND APPROACH

During our research, we plan to apply a design science methodology. First, we identify challenges by systematically considering current research and industrial practices on configuration management. Then, we analysis these challenges to determine, whether we see any potential for improvement on efficiency or effectiveness of the state-of-the-art techniques. Following, we intend to propose novel data structures and algorithms to address the identified challenges or improve existing techniques. For all new methods and approaches, which we propose, we plan to prototypically implement them and evaluate their impact on the respective challenge by comparing them to existing approaches using different real-world SPLs as case studies (e.g., the Linux kernel and BusyBox).

As a platform to implement our prototypes, we rely on the feature-modeling framework FeatureIDE [23, 29, 33]. It is a well-established and matured tool that provides all basic functionalities for managing feature models and configurations and also includes advanced techniques such as several methods for configuration sampling. Furthermore, it is open-source and can be easily extended.

4.1 Threats to Validity

There are some general threats to validity that have to be considered for most of our evaluations. These can be grouped into *internal* and *external* threats, either threatening the validity of results themselves or their generalizability.

Internal. Faults with the implementation of our prototypes can lead to invalid results. We try to mitigate this threat by applying rigorous testing to ensure the sanity of our gathered data. For instance, when we compute a sample for t-wise interaction sampling, we check whether it achieves a 100% coverage for the given value of t and that it does not contain any invalid configurations. Furthermore, we employ matured open-source libraries such as Sat4J [20], FeatureIDE [23, 33], and TypeChef [17] to implement our prototypes.

Within our evaluations, we often rely one some kind of randomness, for instance as a base line or for determining a feature order. A random bias can lead to inaccurate results. In order to mitigate the effects of a random bias, we repeat every affected experiment multiple times.

External. One of our main issues for conducting experiments for an evaluation on automated configuration generation is to find suitable case studies. Currently, it is difficult to find real-world products lines with open source code and a distinct feature model. Often we can get a feature model, but not the code (e.g., due to confidentiality) or we can access the code but not feature model (e.g., because it is hard-coded within the code or stored in an uncommon format). Thus, we often can only rely on SPLs with a rather small feature model or a SPLs that are artificially generated and may not accurately reflect their real-world counterparts. Thus, evaluation results may not be generalizable to all real-world SPLs.

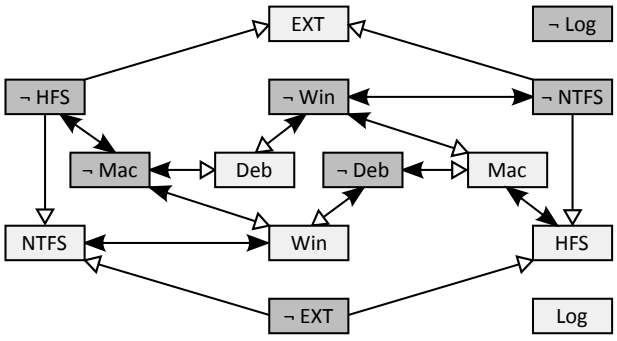


Figure 2: Modal implication graph for *Server* SPL.

5 PRELIMINARY RESULTS

We already made progress on a number of our challenges and also published some of the result at international conferences. In particular, we worked on improving decision propagation for the semi-automatic configuration process (cf. Challenge 1) and improving t-wise interaction sampling for the fully-automatic configuration process (cf. Challenge 3).

5.1 Improving Decision Propagation

Regarding Challenge 1, we presented a novel data structure, called *modal implication graph*, to facilitate decision propagation [19]. A modal implication graph is an extension to regular implication graphs that can store not only pair-wise feature relationships, but arbitrary boolean relationships between three or more features from the feature model.

In Figure 2, we depict a modal implication graph for the feature model presented in Figure 1. Each node represents a state (i.e., selected or deselected) of a feature. In addition there are two types of connections between features. The solid connections are called *strong* edges. If a node is connected to another node via a strong edge, it means that if the starting node is included in a configuration, the end node must be included in the configuration as well. The non-solid connections are called *weak* edges. If the starting node of a weak edge is included in a configuration, there is a possibility that we also have include the corresponding end node. Whether we have to include it or not, is dependent on a more complex relationship that involves other features.

We can utilize the modal implication graph during decision propagation. From the latest decision, we can infer a starting node within the graph (i.e., that most recent selected or deselect feature). From this starting point we can compute other reachable nodes. If a node is reachable via a path that entirely consist of strong edges, it can be directly included in the configuration without further computations. If a node is only reachable via paths that include at least one weak edge, we employ a satisfiability solver to determine, whether it has to be included. For all other nodes, we have to do nothing, saving further calls to the satisfiability solver.

We evaluated decision propagation using our modal implication graph by comparing it to other implementations for decision propagation. We used real-world feature models, including feature models that consist of over 18,000 features. The results show that we can significantly benefit from utilizing a model implication graph during decision propagation. Though it also takes some time to build a graph from a feature model, in our evaluation this one-time cost already amortizes during a single configuration process.

5.2 Improving T-Wise Interaction Sampling

We also worked on finding solutions for Challenge 3. Currently, we achieved two major results. First, we introduced a new pair-wise sampling approach, *IncLing* [1]. With IncLing, we are able to incrementally generate a t-wise sample. That means that we can use the first configurations of the sample while the algorithm is still computing the remaining configurations to achieve the t-wise coverage criterion. This effectively improves efficiency of the algorithm using the sample (e.g., for testing), as we can generate and use configurations in parallel. IncLing also builds on our other work. It utilizes a modal implication graph to determine valid combinations of features.

We evaluated IncLing by comparing it to existing t-wise interaction sampling approaches. As case studies, we used different real-world feature models. We found that while it does not always compute the smallest sample it is often significantly faster than other algorithms.

Second, we introduced an alternative to t-wise interaction sampling, t-wise presence condition sampling. For t-wise presence condition sampling, we do not only consider features in the feature model, but also the actual variability implementation. For each implementation artifact of an SPL, we can determine a presence condition, that is a combination of features, for which this implementation artifact is included in a product. Instead of considering interactions between t features, we are considering the interactions of t presence conditions. With this technique we aim to achieve a better code coverage, and thus, a more effective sample. We originally introduced t-wise presence condition sampling for testing purposes, thus we consider effectiveness in terms of possibly detected faults. A paper for this contribution is currently under review. Similar to IncLing, this technique utilizes modal implication graphs as well.

We conducted an evaluation of t-wise presence condition sampling by comparing it to other t-wise interaction sampling techniques. In order to measure effectiveness, we used mutation testing from previous work [32]. We found that we can achieve a similar effectiveness as the other evaluated algorithms. However, we also found that on average the samples produced by t-wise presence condition sampling contain substantially less configurations indicating, that, we can improve efficiency on t-wise sampling using this novel technique.

6 WORK PLAN

Building on our already achieved results, our further work plan for the thesis is as follows. Within the next year, we plan to work on extending and improving modal implication graphs and present a novel technique to generate uniform random samples. Beyond that, we want to address the remaining Challenge 4 for feature model evolution that we did not consider thus far.

Currently, model implication graphs work quite well for decision propagation and other analyses. However, there are some potential

improvements and extensions we want make. First, it is hard to utilize the modal implication graph for revoking previously made decisions in the configuration process. In this case, we cannot use the full potential of the graph and may miss some potential for reducing calls to the satisfiability solver, as the graph's current layout only work well in the other direction. This contribution again addresses Challenge 1.

Second, we plan to find a way to efficiently update the graph if the feature model evolves (cf. Challenge 4). At the moment, we have to recompute the graph in case of a change to the feature model, which increases the computational overhead. However, we aim improve this by considering the actual change on the feature model and updating the graph accordingly by only adding or removing single edges or nodes. We intend to submit the results to an international journal, such as IEEE Transactions on Software Engineering (TSE), ACM Transactions on Software Engineering and Methodology (TOSEM), or the International Journal on Software and Systems Modeling (SoSyM).

In addition, we want to focus on Challenge 2 and improve uniform random sampling. We intend to present a technique that implements approximate uniform random sampling. With such a technique we aim to be able to achieve nearly-perfect uniformly distributed samples but with much less computation time required. We plan to submit this contribution to an international conference on software engineering, such as Conference on Systems and Software Product Lines (SPLC) or the International Conference on Software Engineering (ICSE).

Further in time, we want to consider feature model evolution in more detail and its influence on existing configurations. Depending on the novelty and significance of our contribution, we consider either submitting to a journal or international conference on software engineering.

ACKNOWLEDGMENTS

This paper was supported by the German Research Foundation under GrantNo.: LE 3382/2-3.

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 *Proceedings of the International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 144–155.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[3] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*. Springer, 7–20.

[4] Don Batory, David Benavides, and Antonio Ruiz-Cortés. 2006. Automated Analyses of Feature Models: Challenges Ahead. *Commun. ACM* (2006).

[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), 615–708.

[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 *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. ACM, 7:1–7:8.

[7] 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 International Conference on Automated Software Engineering (ASE)*. ACM, 73–82.

[8] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. 2015. Reasoning about Product-Line Evolution Using Complex Feature Model Differences. *Automated Software Engineering* 23, 4 (2015), 687–733.

[9] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley.

[10] Krzysztof Czarnecki and Andrzej Wąsowski. 2007. Feature Diagrams and Logics: There and Back Again. In *Proceedings of the International Software Product Line Conference (SPLC)*. IEEE Computer Science, 23–34.

[11] José A Galindo, Mathieu Acher, Juan Manuel Tirado, Cristian Vidal, Benoit Baudry, and David Benavides. 2016. Exploiting the enumeration of all feature model configurations: a new perspective with distributed computing. In *Proceedings of the 20th International Systems and Software Product Line Conference*. ACM, 74–78.

[12] Christopher Henard, Mike Papadakis, and Yves Le Traon. 2014. Mutation-Based Generation of Software Product Line Test Configurations. In *Search-Based Software Engineering*, Claire Le Goues and Shin Yoo (Eds.). Lecture Notes in Computer Science, Vol. 8636. Springer International Publishing, 92–106.

[13] 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 (TSE)* 40, 7 (2014), 650–670.

[14] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. 2012. A User Survey of Configuration Challenges in Linux and eCos. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. ACM, 149–155.

[15] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. 2012. A User Survey of Configuration Challenges in Linux and eCos. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. 149–155.

[16] Mikolas Janota. 2008. Do SAT Solvers Make Good Configurators?. In *Proceedings of the International Software Product Line Conference (SPLC)*. 191–195.

[17] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proceedings of the International SPLC Workshop Feature-Oriented Software Development (FOSD)*. ACM, 25–32.

[18] 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 *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE)*. ACM, 291–302.

[19] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. 2018. Propagating Configuration Decisions with Modal Implication Graphs. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 898–909.

[20] Daniel Le Berre and Anne Parrain. 2010. The Sat4j Library, Release 2.2, System Description. *Journal on Satisfiability, Boolean Modeling and Computation* 7, 2-3 (2010), 59–64.

[21] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE)*. ACM, 81–91.

[22] Sascha Lity, Sophia Nahrendorf, Thomas Thüm, Christoph Seidl, and Ina Schaefer. 2018. 175% Modeling for Product-Line Evolution of Domain Artifacts. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. ACM, 27–34.

[23] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.

[24] Marcílio Mendonça. 2009. *Efficient Reasoning Techniques for Large Scale Feature Models*. Ph.D. Dissertation. University of Waterloo.

[25] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-Based Analysis of Feature Models is Easy. In *Proceedings of the International Software Product Line Conference (SPLC)*. Software Engineering Institute, 231–240.

[26] 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 (TSE)* 41, 8 (2015), 820–841.

[27] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. 2018. Anomaly Analyses for Feature-Model Evolution. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 188–201.

[28] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding near-optimal configurations in product lines by random sampling. In *Foundations of Software Engineering*. ACM, 61–71.

[29] Juliana Alves Pereira, Sebastian Krieter, Jens Meinicke, Reimar Schröter, Gunter Saake, and Thomas Leich. 2016. FeatureIDE: Scalable Product Configuration of Variable Systems. In *Proceedings of the International Conference on Software Reuse (ICSR)*. Springer, 397–401.

[30] Juliana Alves Pereira, Pawel 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.

[31] 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 *IEEE Conference on Software Testing, Validation and Verification (ICST)*. 240–251.

[32] Sebastian Ruland, Lars Luthmann, Johannes Bürdek, Sascha Lity, Thomas Thüm, Malte Lochau, and Márcio Ribeiro. 2018. Measuring Effectiveness of Sample-Based Product-Line Testing. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 119–133.

[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 (SCP)* 79, 0 (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 International Software Product Line Conference (SPLC)*. ACM, 1–13.

Energy Efficient Assignment and Deployment of Tasks in Structurally Variable Infrastructures

Angel Cañete
Universidad de Málaga
Andalucía Tech, Málaga, Spain
angelcv@lcc.uma.es

ABSTRACT

The importance of cyber-physical systems is growing very fast, being part of the Internet of Things vision. These devices generate data that could collapse the network and can not be assumed by the cloud. New technologies like Mobile Cloud Computing and Mobile Edge Computing are taking importance as solution for this issue. The idea is offloading some tasks to devices situated closer to the user device, reducing network congestion and improving applications performance (e.g., in terms of latency and energy). However, the variability of the target devices' features and processing tasks' requirements is very diverse, being difficult to decide which device is more adequate to deploy and run such processing tasks. Once decided, task offloading used to be done manually. Then, it is necessary a method to automatize the task assignation and deployment process. In this thesis we propose to model the structural variability of the deployment infrastructure and applications using feature models, on the basis of a SPL engineering process. Combining SPL methodology with Edge Computing, the deployment of applications is addressed as the derivation of a product. The data of the valid configurations is used by a task assignment framework, which determines the optimal tasks offloading solution in different network devices, and the resources of them that should be assigned to each task/user. Our solution provides the most energy and latency efficient deployment solution, accomplishing the QoS requirements of the application in the process.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; • **Computer systems organization → Distributed architectures**; *Embedded and cyber-physical systems*; • **Hardware → Power estimation and optimization**.

KEYWORDS

Software Product Line, Mobile Edge Computing, Mobile Cloud Computing, Energy Efficiency, Latency, Optimisation

ACM Reference Format:
Angel Cañete. 2019. Energy Efficient Assignment and Deployment of Tasks in Structurally Variable Infrastructures. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3307630.3342704

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342704

1 INTRODUCTION AND MOTIVATION

The importance of cyber-physical systems (CPS) [24] continues growing, due to the massive popularization of mobile devices for personal use (smartphones, personal devices, tablets, etc.), the improvement in the communication speed of wireless networks, the development of the Internet of Things (IoT) [1] and the benefits of Cloud Computing (CC) [29], which has played an important role in the development and advancement of mobile applications for the IoT. More recently, technologies such as Edge Computing [38] and Fog computing [5] bring the advantages and power of the cloud closer to where the data is created and consulted.

It is expected that by 2025 there will be around 80 billion of IoT devices -from personal mobile devices, to sensors, household appliances and wearables- connected to Internet, which would generate 175 trillion Gb of data per year [35]. In return, these forecasts imply an increase in traffic in the network that cannot be assumed by the cloud.

Edge Computing (EC) is a decentralized computing infrastructure in which data, computation, storage and applications are located somewhere between the source of the data and the cloud. This technology aims to improve the efficiency of the infrastructure, transferring the most hard-computational tasks of applications to any nearby device with network connectivity and computing power [27]. By this way, the data can be processed or stored nearby it is produced and/or consumed. Among the advantages of EC we can include the reduction of the communication latency respecting to send the data to the cloud (Mobile Cloud Computing, MCC) [41], the reduction in the energy consumption that can be achieved, and others benefits, like the data privacy obtained. EC takes advantage of the large amount of inactive computational power and distributed storage space in different Edge devices (such as Wi-Fi routers and access points, or low-cost computer boards - SBC) located nearby final users and devices (called edge of the network), and connected to the same WLAN. These devices, that forms the deployment infrastructure (DI), perform computationally intensive and/or critical latency tasks. This is known as Multi-Access Edge Computing [33] (formerly Mobile Edge Computing, MEC). These advances in TIC technologies influence the way in which applications are implemented and deployed in the IoT devices, the edge or the cloud [40, 46].

Features of the devices that conform the DI (called *nodes*) are very diverse: computing power, memory capacity, software characteristics, storage, communication latency, etc. Due to this variability,

it is not easy to know which node is the best option to deploy applications' tasks to obtain the benefits that MEC and MCC can provide, and accomplishing the QoS requirements of the applications in the process. It is necessary the creation of new methods and software tools that help developers to manage the variability of the hardware, the complexity of network access infrastructures, the distribution of data, and the decision of where the different computational tasks must be located. These methods and tools would help them in the applications' configuration and deployment, obtaining a good QoS level in the process (in terms of latency, security, etc).

A widely number of works propose code offloading as a solution to improve the applications' energy consumption and latency [14, 16, 19, 20, 23, 25, 28, 30, 34, 40, 43, 44, 49, 50]. Nevertheless, none of them give a solution to model and configure both applications and deployment infrastructures. In addition, only a few apply MCC and MEC at the same time [44, 49], while others do not take into account some factors of the process. For instance, some approaches do not considerate the execution time of tasks offloaded to the cloud [12, 28, 47, 48], while others ignore the communication time to obtain the response from the cloud [11], or just take into account the energy consumption of the CPU [4, 50]. Several MEC based approaches do not consider the workload of the edge devices [28, 37]. In this thesis, we will take into consideration all these issues, providing an engine to weight them according to the company/organization interests.

The main contributions of our proposal are the following:

- **We will create a model to configure DIs and applications for code offloading:** In order to obtain the optimal application deployment solution, it is primordial to considerate both the heterogeneity of applications and nodes of the DIs. Applications are composed by a set of configurable tasks. In the same application domain, every application can be configured to meet different requirements. At the same time, the energy consumption and latency of the applications' tasks will depend on hardware (e.g., node's architecture, CPU power, transmission power, etc) and software features (e.g., data to be transmitted, tasks' computational cost, etc) among other issues. This variability present in tasks and nodes can be modeled using SPL (Software Product Lines) [32] feature models, which allow the configuration of the DIs and the applications to meet specific deployment scenarios. In this thesis we will propose a general model for deployment infrastructures in edge and cloud, and will focus on the family of Augmented Reality (AR) [2] applications. We will decompose the functionality of this applications in intensive computational tasks that could be offloaded to the edge or cloud, which are characterized by the required resources and time requirements.
- **Our approach will define a process to generate the optimal deployment solution according to the application and the DI at runtime:** The configuration of the DI and the application variability models is the input of the Optimal Task Assignment Framework (OTAF), which resolves what is the optimal deployment solution in terms of energy consumption and latency. At the same time, the OTAF will assign hardware resources (storage, RAM, and CPU) to each user. The functional and non functional requirements of the application are taken into account during the process (e.g., hardware requisites, QoS needed, security requirements, etc). To achieve this, the problem must be formalized and solved. Data structures based on weighted graphs, and solver techniques like graph cutting or linear and nonlinear programming can help us in this task [27]. Since the OTAF is a daemon, and it is running in a node (manager node) continuously, each time a new user starts the application, it returns the optimal deployment solution and resources assignment according to the current state of the DI. Finally, to put into the practice this approach, we plan to make use containers [39], virtualization [26] or unikernel [6] solutions.
- **We will provide a solution to adapt user's applications at runtime:** The functionality of the application carried out by the device of the user, typically a smartphone, will depend on the deployment solution. Although the application's behaviour would change from one execution to another, due to the deployment assignment at the nodes of the DI, the application itself will be the same. It makes necessary the introduction of a mechanism to reconfigure applications according to this. For this reason, we will provide an adaptation engine for smartphones. As this can be done by different ways, we will implement a few solutions to adapt applications at loadtime and runtime. Then, we will compare them in order to find the most energy-efficient adaptation engine for smartphones.

The rest of the paper is organized as follow: Section 2 clearly states the research questions that we try to answer in this thesis. Section 3 shows the methodology and our approach. In Section 4 we present the preliminary results. Finally, in Section 5 we outline the structure of this thesis, the actual state of it and a year work plan.

2 RESEARCH QUESTIONS

As part of my thesis, our approach has to answer several research questions in order to fulfill the aim of this project.

Firstly, since we need information about the application's tasks as well as the DI's nodes they will be deployed on, the first question we would like to answer is: **RQ1: How can we model the variability of DI and the application's tasks in order to optimize the deployment assignment solution?** In this thesis we will bring a model to configure DIs that could be applied to any scenario. Nodes of the DI are composed by a set of features that describe them (CPU power, RAM, operative system, kind of device, etc). To model all these features, we will use SPL feature models. Specifically, we will use features with numerical values [7]. Our deployment infrastructure will be composed by a set of nodes. For this reason, we use cardinality-based feature models [17]. As testbed application, we will focus on AR applications' family. Many mobile devices are often unable to cope with the runtime real time requirements of these multimedia apps. Offloading to the cloud is not always a feasible option due to the high and inconstant latency of wide-area networks [42]. In any case, we need to determine which information must be contained in the models in order to determine

the energy consumption of the deployment and the maximum latency permitted. The information resulting by the configuration of the application and the DI feature models is used as input of our assignment problem. So, **RQ2: How can be the problem formalized and solved?** The assignment of tasks and resources to the nodes of the DI is not an easy process. Functional requirements (e.g., the application need a camera to run) as well as non functional requirements (e.g., QoS, security, etc) must be taken into account. In order to accomplish the QoS, dependencies between tasks must be considered. These dependencies can be of two types: sequential, where one task ends before the other task begins, or parallel, where it is not necessary that the previous task ends before starts the next one. Some tasks (or a set of them) could have time limitations in order to achieve the QoS requirements. We need a model that allows us the management of tasks dependencies, along with time restrictions. Graph representation allows both kind of dependencies [22], and linear and/or nonlinear programming are positioned as good ways to solve our problem.

Tasks' assignment process is not only difficult, but it is also a hard-computational process. Therefore, we wonder if it it possible to do it at runtime, and **RQ3: How can we afford a deployment assignment at runtime?** The deployment solution obtained from RQ2 will be provided by a microservice architecture, *structuring the application as a collection of services in order to distribute the tasks that compose the application among the nodes of the DI.* This service must be able to bring a solution at loadtime in a time short enough to be invisible to the final user. Due to the problem's complexity, it could be difficult to achieve this goal. For this reason, collecting information about previous executions allow us to predict the optimal solution for the current problem. As smartphones may not be very resource-rich devices, our service must be running on a DI's node, called manager node.

RQ4: How can be the tasks deployed on DI's nodes? Once an optimal deployment solution is found, we need to automatically deploy each task to the assigned node. These nodes should distribute and reserve part of their resources to each task, so it is necessary a mechanism to make it possible. During the execution, it might happen that some nodes stop working. In addition, the location of the user may change along the execution, being some nodes inaccessible from the new location. Nevertheless, the service cannot be interrupted. For this reason, the solution must include data replication (fault tolerance) and data and functionality migration at runtime from one node to another without stopping the application, making as result a self-adaptive system able to change the tasks' allocation if necessary. Some existing technologies, such as virtualization, containers (e.g., Docker) or unikernel solutions, facilitate redeployment. We will explore all of them in terms of energy consumption, strengthness and weakness , and their influence in QoS. When all offloaded tasks from the user's node to the DI are running, the application of the user must be adapted in order to execute only the tasks that have been assigned to it. We focus in scenarios where user' nodes are smartphones. It poses the next question: **RQ5: How can applications being adapted to the edge-based deployment assignment?** We need a solution to adapt existing applications to the deployment assignment. As the majority users have Android smartphones, we will focus our approach in Android applications, although we do not discard to present a solution to adapt iOS applications too. Our adaptation approach must be as lightweight as possible, consuming a little amount of energy. To achieve this, we will propose different adaptation engines (e.g., based on proxy pattern [21], virtual-method hooking [15]) and then we will measure the energy overhead introduced by them, as well as their scalability. Energy consumption in Android smartphones can be measured using different existing software tools, such as Trepn Profiler[1] or GreenScaler [13].

Once the application is deployed and it is running, we have to determine how good is our deployment assignment. **RQ6: What is the benefit obtained by our optimal assignment process?** Minimize the applications' energy consumption and latency are the goals of this thesis, satisfying the functional and non functional requirement on the way. On one side, we need a validation module that verifies that these requirements are accomplished (functional testing, execution time tests, security testing, etc). On the other side, the benefit obtained by our solution should be evaluated. It can be done by measuring the execution time and energy consumption of the code offloading solutions and then comparing it with the result of execute all the application's tasks at the node of the user. Measuring the energy consumption is not an easy task, as it is highly dependent on the architecture of the device (hardware and software features), its state (e.g., workload), and environmental conditions. Some of them, like the background processes running on the devices, could be difficult to control.

3 RESEARCH METHODOLOGY AND APPROACH

For the accomplishment of the objectives presented in this thesis, we will use a SPL based approach to model the variability of applications and DIs (RQ1). The optimal assignment deployment problem must be formalized, in order to use the information obtained by the models to bring the optimal deployment solution (RQ2).

Our research in this thesis is both applied and empirical. We plan to develop a service-oriented solution to, given a configured deployment infrastructures and application, to generate the optimal deployment assignment (RQ2, RQ3). The tasks are offloaded from the user's node (RQ5) to be deployed to different nodes of the DI (RQ4). In order to obtain the optimal deployment assignment, we plan to use both linear solvers (like Z3) [18] and nonlinear solvers (like Gekko) [3]. The framework will run as a microservice in a manager node and will be attending requests from users. The manager node contains information about the state of each node of the DI (hardware and software information, network and workload). The manager node is responsible for sending the instructions to the rest of nodes in order to deploy application's tasks, as well as assigning resources to each task. We will define the mechanisms to maintain the manager node updated with the status of the rest of nodes, and a protocol to control and perform tasks deployment and data migration. Nodes could use virtualization, containers or unikernel based solutions. At the end, the application works like a microservice-based architecture, where the application's tasks are distributed into different nodes. All the process must be approved by a validation and verification module, which verifies that both

[1] https://developer.qualcomm.com/software/trepn-power-profiler

functional and non functional requirements are accomplished, as well as evaluates how good is the solution (RQ6).

Figure 1 shows a full view of our approach. One part of our proposal is based on domain engineering methodology (top of Figure 1, Sections 3.1, 3.2 and 3.3) and the other part is based on application engineering methodology (bottom of Figure 1, Sections 3.4, 3.5 and 3.6).

3.1 Model of the Deployment Infrastructures

The set of nodes that compose the DI are modeled using cardinality feature models with numerical values. On top of Figure 1, the SPL model of a DI is shown. In this model, nodes have a type associated: they can be computational nodes, user nodes or sensor motes. Computational nodes are devices in which tasks of the user can be offloaded in order to save energy or time. For example, computers, smartphones, IoT Gateways or limited resources computers, like Raspberrys. User nodes are devices in which the main application is executed. Once a user node is connected to the service, it can form part of the DI like computational node or not, according to the policy. By last, sensor motes are nodes in a sensor network capable of performing some processing, gathering sensory information and communicating with other connected nodes in the network. The type of information collected by sensor motes depends on the sensor units associated to them. In the model shown on top of Figure 1, there is a constraint that makes mandatory for sensor motes and smartphones to have at least one sensor unit associated. Some application's tasks require one or more sensor units to be executed, so this allows us to determine which nodes are able to run each task (e.g., the task *CaptureVideoFrame* must be ran in a node with an image sensor unit associated). The nodes are composed by a set of numerical features that describe them (CPU power, HD capacity, RAM). Finally, the nodes are connected to internet using a network with a numerical feature associated, its bandwidth, that describes the data transmission rate of the node.

3.2 Model of the application's family

An application's family is composed by a set of software applications that share the major of their features. This allows the configuration of a collection of applications and the reuse of software components. The feature model of an application must contain the information necessary to describe it. The configuration of the feature model of an application determines its functionality, which is used to select the set of tasks that form the application. These tasks have software and hardware requirements that are modeled in the feature model and their granularity depends on the dependency between their functions.

3.3 Optimal Tasks Assignment Framework

The information obtained by the configuration of the DI and the application is used by the OTAF to assign tasks to DI's nodes. The assignment must accomplish both, functional and non functional requirements of the application. It is necessary to define a structure of data that allows to manage the information of tasks (e.g., dependencies, time restrictions, etc) and a solution to convert the data from the feature models to the OTAF's format. Graph representation is posed as a good technique as it allows to manage tasks features and connections between tasks. The OTAF needs information about tasks that are not provided by the feature model of the application, like tasks dependencies or the sets of tasks with time restrictions. This kind of information is pre-planned in the OTAF for the possible input values of the applicatons' configuration.

We will formalize the assignment process like an optimization problem with restrictions, with the objectives of minimizing the energy consumption and latency. Time and energy consumption due to tasks' computation and communication must be defined in order to predict the execution latency and energy consumption of the deployment assignment. We will define polices to center the energy profit on different parts of the DI (e.g., user node, battery powered nodes of the DI, DI's overall powered consumption, etc). Like solvers, we will start using Z3 (for linear optimization problems) and Gekko (for nonlinear optimization problems). Nevertheless, we do not discard the use of other kind of solutions based on genetic algorithms, for example.

The Optimal Tasks Assignment Framework must return the optimal solution in a short time, making possible the assignment at runtime (RQ6). The potential application areas where IoT and edge computing solutions are advantageous, help to show the contributions and benefits of our approach (e.g., eHealth applications [31], smart campus, smart buildings, etc). For instance, in an eHealth application that provide its service in an hospital, the number of users and nodes makes the problem size considerable. One of the main motivation for the decentralization promoted by EC is to support IoT infrastructure scalability rather than mobile applications' interactive performance. The scalability of our proposal must be evaluated in order to predict its applicability for different sizes of the problem and scenarios. The number of assignments for tasks and nodes is $nodes^{tasks}$, so for a large number of nodes and tasks the OTAF may need too time to provide the solution at loadtime. Nevertheless, it can be done by collecting information about previous executions and processing it in order to predict the most appropriate solution for new users (e.g., by using machine-learning techniques). As we focus on mobile users (battery powered devices), the battery level of the user's device is a crucial information to select the policy of the deployment assignment (e.g., maximum power saving on user node). By last, the service of the OTAF must be running on a DI's node (manager node). It can be implement like a REST API, running on a Nginx server[2].

3.4 Tasks' deployment

The solution obtained by the OTAF must be deployed on the DI. In order to do that, we explore solutions based on unikernel, virtualization, and containers. At first look, VM virtualization is more heavyweight than the others [45], as its emulates the hardware of the computer. Containers, instead of virtualizing the underlying computer like a virtual machine (VM), just the OS is virtualized. By last, unikernel solutions pop the OS from the execution stack, compiling all the necessary (including operating system support functions) to run the application in a single executable. Unikernel solutions make the system faster, smaller, and safety; as the OS is deleted from the execution stack, its vulnerabilities are removed

[2] http://nginx.org/en/

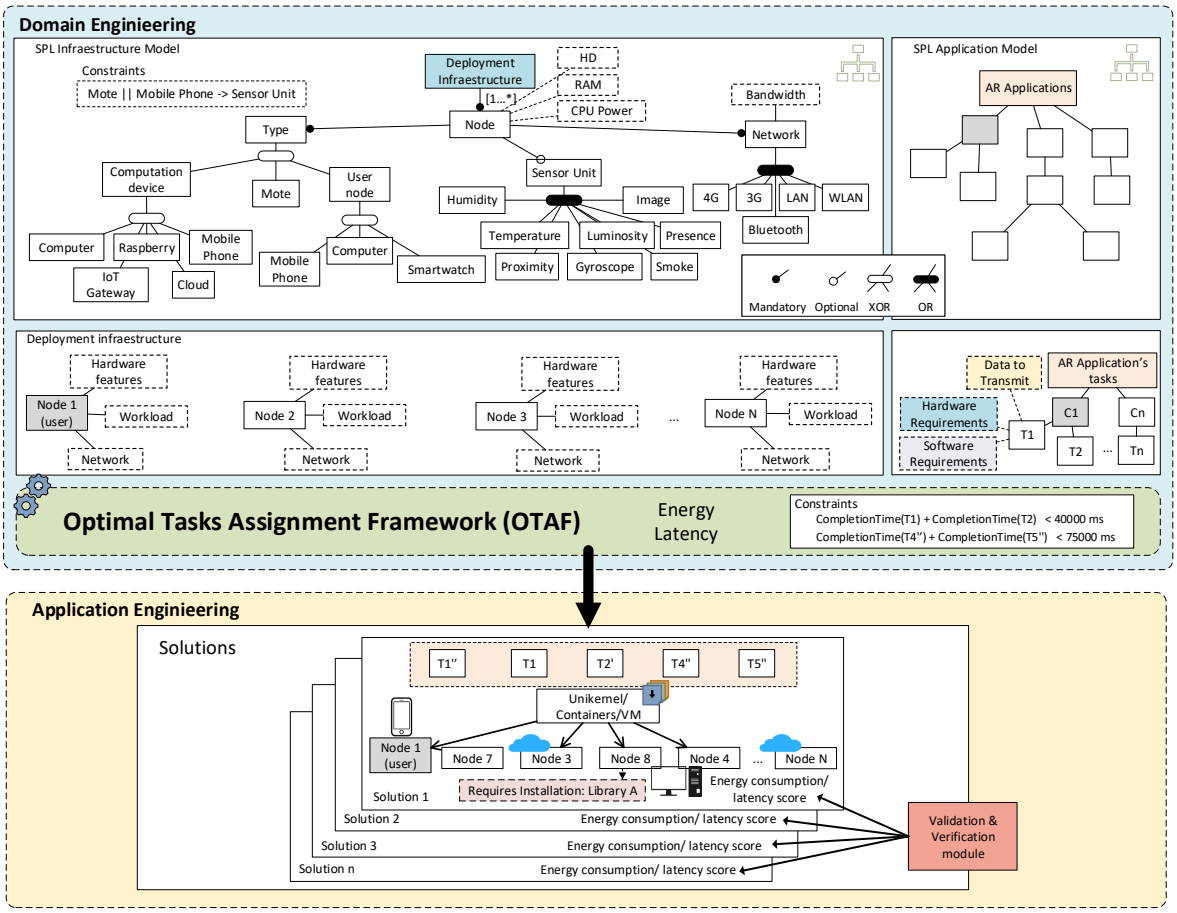


Figure 1: Our approach

too. For a part of the application to start, all the parts of the application that it depends on must be ready. These dependencies between parts of the application, whatever the deployment solution (unikernel, virtualization or containers based), will be evaluated based on ready conditions.

During the execution of the application, it is possible that some DI nodes stop working. Our system must be able to detect faults in the DI's nodes and fix them without stopping the service provided to the user. In order to do that, the manager node maintains information about the state of the DI's nodes and make a backup of the tasks' executions periodically. This information may be sensible and can contain private data; taking into account the decentralized nature of Edge Computing, it will be encrypted in order to secure it. If the manager node detects that a node has stopped, the tasks that were running on this node are assigned to other operational nodes. These nodes are notified in order to deploy the assigned tasks and receives the information of the users executions, obtained by the manager node by restoring it from its backup. Changes in the DI's nodes affect to the prediction system, which has to take into account only the available nodes. In the same way, the service provided by the manger node must be replicated in order to provide fault tolerance in this node.

3.5 Adaptation engines for the applications

The behavior of the application running on the user's device will depend on the deployment assignment at the DI's nodes. In this thesis, we focus on AR applications, where the node of the user is typically a smartphone. Adaptation engines based on dynamic proxies and method hooking (e.g., using Xposed framework [36]) will be explored in order to adapt the applications' functionality to the deployment assignment. Several adaptation engines will be implemented and compared between them. Our aim is to obtain the most energy-efficient adaptation engine at runtime/loadtime. We will focus on Android OS because it is the most extended mobile platform, although we do not discard its research for other mobile OS.

3.6 Verification and validation module

The first step to verify the deployment assignments is to check if software and hardware requirements are accomplished. It can be done by a feed-backing between the configuration of the DI and the application. Then, functional testing (using unit test cases) is necessary in order to prove that the application works as expected. In terms of safety, we will focus on data protection and application access. Tools like BeEF[3] could help us for this task.

[3]BeEF: https://beefproject.com/

For the purpose of knowing how good is our solution, energy consumption and latency must be measured. It can be done by using software or hardware based tools. Software tools, like pTopW [10] (Windows), PowerTop[4] (Linux) or Trepn Profiler and GreenScaler (Android) allow the estimation of the energy consumption via software. Alternatively, hardware based tools (like Kill A Watt[5]) provide more accuracy and are OS independent. Nevertheless, it has no sense its usage in battery powered devices.

3.7 Threats to Validity

As we move forward in our research: (1) the number of tasks and nodes can make the problem too hard-computational to be solved at runtime; (2) although we can estimate the energy consumption of the application's tasks in each node of the DI, it is not possible to know the exact energy consumption of them in the user's node (hardware and software variability, background processes, etc); (3) the amount of energy saved will depend on the DI's nodes, as well as the number of tasks that the user will be able to offload to them; (4) in unikernel based solutions, CPU resources are assigned to users in terms of CPU cores available, hindering the prediction of workload of the nodes in each moment to accomplish the QoS requirements.

4 PRELIMINARY RESULTS

So far, I have presented a full paper [9] at "Jornadas de Ingeniería del Software y Bases de Datos" (JISBD 2018). In this paper, four different adaptation engines for Android applications' code are discussed, in order to find the most energy-efficient solution: (1) based on dynamic proxies, where the adaptation alternatives (functions that replaces the default behavior of the application) are internal to the application; (2) based on dynamic proxies, where the adaptation alternatives are external to the application (e.g., placed in device external storage, placed on a server, etc); (3) method-hooking and proxies based; and (4) based on method-hooking. As scenarios in which these engines would be used are very diverse (e.g., number of adaptation alternatives, number of rules that determines if adaptations should be triggered, etc), the scalability of the adaptation engines is evaluated. It is demonstrated that the number of adaptable functionalities (classes), the number of adaptation rules and the number of adaptation alternatives do not increase the energy consumption of the application. The impact in the energy consumption by the engines themselves are evaluated, concluding that the energy consumption of the isolated adaptation engines is tiny compared with the application one, that is of the 2.68% in the worst case. It makes our solution a good candidate to be applied to the schema presented in this thesis, in order to adapt the application running in the user's device to the application deployment solution at the DI's nodes. Some solutions presented in [9], like (1) and (2) could be used in others OS like iOS, as they are based on a design pattern.

Currently, I am working on a project that forms the basis of this thesis, presenting a first version of the DI feature model, as well as a feature model for AR applications' family. In this work, we formalize the Optimal Assignment Problem like a Linear Optimization Problem. Then, the problem is solved using the Z3 solver. Finally, the latency performance of the solver for different sizes of the problem is evaluated. A first version of this work [8] will be presented at JISBD 2019. In this paper, the first version of the OTAF is presented and evaluated. The Optimal Assignment Problem is posed like mono-objective, where the aim is to optimize the energy consumption taking into account the functional and non functional requirements of the applications. The service of the OTAF is provided in a node with a AMD Ryzen 7 1700X processor, using one core for compute the assignment problem. In this case, the system is able to bring a solution in 0.76 seconds for a application composed by 11 tasks and a DI formed of 5 nodes, obtaining a benefit of the 58% in the energy consumption compared with the execution of the application in the user node.

5 WORK PLAN

This thesis has two main edges; a component based on SPL and a part based on application engineering. First, we define the SPL component (Task 1, 2) and then we apply application engineering to use the information provided by the SPL component to deploy the application's tasks among the DI's nodes (Tasks 3,4,5, and 6). The results of Task 6 give us a feedback information which will be used to improve the model (Task 1, 2). Figure 2 shows a twelve-month work plan. Concretely, the tasks are the following:

Task 1: Refine and improve the feature models. Until now, we have developed a first version of the variability models of AR applications' family and DI. Nevertheless, we need to continue improving them in order to achieve complete and accurate variability models. This task will continue till the middle of July.

Task 2: Assign CPU resources in the nodes of the DI to users. Initially in parallel with Task 1, and further till end-August, we will work in the assignment of the CPU resources of the DI's nodes to users. For this task, we need to formalize and implement the Optimal Assignment Optimization Problem as a nonlinear programming problem. The first choice is to use the Gekko solver, although we do not discard to use another nonlinear solver or even the usage of other kind of solutions for optimization, such as, for instance, solutions based on genetic algorithms. As result, we will obtain a first version of the OTAF with CPU resources assignment.

Task 3: Implementation of the AR applications' tasks as microservices. Tasks of AR applications that can be offloaded to DI's nodes for their execution will be provided as microservices. So, the first step is to define which tasks of AR applications' family can be offloaded to nearby nodes. This step may involve having to change the variability model of the application from Task 1. Once tasks are clearly defined for offloading to the DI, their functionality must be implemented. Finally, these functionalities will be provided by a microservice architecture. In order to do that, we will explore different techniques (e.g., RESTful API, SOAP). We will work in this task from start September to the middle of November.

Task 4: Provide the OTAF like a microservice. From the beginning of November to the middle of December, we will work to supply the functionality of the OTAF (Task 2) like a microservice in

[4]PowerTop: https://01.org/powertop/
[5]Kill A Watt: http://www.p3international.com/products/p4400.html

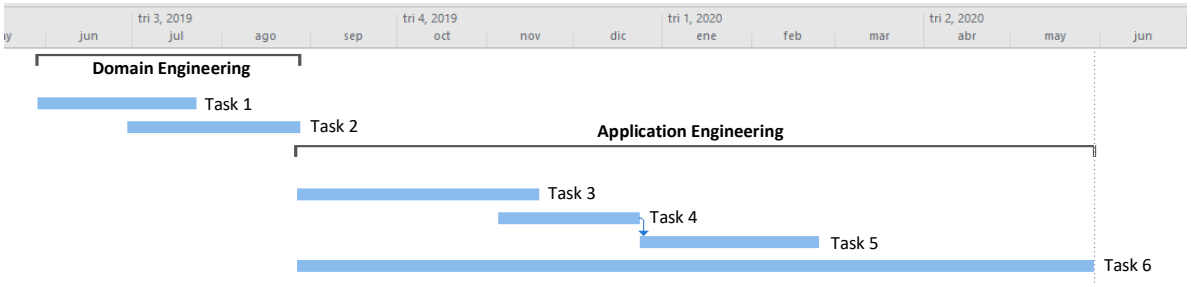


Figure 2: One year work plan

one node of the DI (manager node). This node contains information about the state of all nodes of the DI and provides the service to assign one node to run each task of the users' applications. The creation of the OTAF microservice is a crucial part of this thesis. This microservice can use all the resources of the manager node or only part of them, running on a VM or a container (e.g., Docker).

Task 5: Implementation of the deployment process for AR applications. From the middle of December to the middle of February 2020, we will put into practice the optimal assignment solution obtained by the OTAF (Tasks 2 and 4). Application's tasks have to be allocated at the nodes of the DI according the task assignment, using the minimum amount of resources in the process. For this purpose, the use of virtualization, containers and unikernel solutions will be explored.

Task 6: Tasks' deployment evaluation. This task, which evaluates the energy consumption and execution time, will start at the beginning of Task 3 and will continues after the end of Task 5. At its beginning, during Task 3, Task 6 evaluates the deployment of the each task of the AR applications and other case studies considered. During Task 4, the energy consumption and execution time of the OTAF's microservice will be evaluated, and during Task 5, the evaluation will involve the procedure of assignment of tasks as a whole. From the analysis result, some corrections can be made in the feature model to improve the correctness of the model (Task 1). Quality of deployments will be measured, evaluated and compared in terms of energy consumption and latency. This task will continue to the end of May.

5.1 Publication plan

The form in which the problem is structured poses a set of challenges that should be accomplished separately. All the solutions for these challenges allow us to obtain intermediate solutions susceptible to be published in international congresses and workshops. Due to the features of the thesis, the edges of the conferences and workshops can be diverse, like based on Software Product Lines, microservices or IoT services, according to the topic of the specific approach. The most extended works will be sent to journals indexed in JCR.

Some journals, conferences and workshops appropriated to distribute our solutions are:

- **Journals:**
  - Information and Software Technology
  - Empirical Software Engineering
  - Journal of Systems and Software
  - Software and Systems Modeling
  - Journal of Universal Computer Science
  - Information Systems Research
  - ...
- **Conferences and Workshops:**
  - International Conference on Mobile and Ubiquitous Systems: Networks and Services (MobiQuitous)
  - Workshop on Mobile Computing Systems and Applications (HotMobile)
  - Distributed Applications and Interoperable Systems (ICWS)
  - Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)
  - Systems and Software Product Line Conference (SPLC)
  - European Conference on Software Architecture (ECSA)
  - International Conference on Software Reuse (ICSR)
  - IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS)
  - ...

ACKNOWLEDGEMENTS

This work is supported by the projects Magic P12-TIC-1814, HADAS TIN2015-64841-R (co-funded by FEDER funds), TASOVA TIN2017-90644-REDT, and MEDEA RTI2018-099213-B-I00 (co-funded by FEDER funds).

REFERENCES

[1] Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A survey. *Computer Networks* 54, 15 (2010), 2787 – 2805. https://doi.org/10.1016/j.comnet.2010.05.010

[2] Ronald T Azuma. 1997. A survey of augmented reality. *Presence: Teleoperators & Virtual Environments* 6, 4 (1997), 355–385.

[3] Logan D. R. Beal, Daniel C. Hill, R. Abraham Martin, and John D. Hedengren. 2018. GEKKO Optimization Suite. *Processes* 6, 8 (2018). https://doi.org/10.3390/pr6080106

[4] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. 2012. Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing. *Future Generation Computer Systems* 28, 5 (2012), 755 – 768. https://doi.org/10.1016/j.future.2011.04.017 Special Section: Energy efficiency in large-scale distributed systems.

[5] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing (MCC '12)*. ACM, New York, NY, USA, 13–16. https://doi.org/10.1145/2342509.2342513

[6] A. Bratterud, A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. 2015. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. 250–257. https://doi.org/10.1109/CloudCom.2015.89

[7] Rafael Capilla and Juan C. Dueñas. 2002. Modelling Variability with Features in Distributed Architectures. In *Software Product-Family Engineering*, Frank van der Linden (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 319–329.

[8] Angel Cañete, Mercedes Amor, and Lidia Fuentes. 2019. Sistema de Asignación de Tareas Energéticamente Eficiente en Infraestructuras de Despliegue Variables. (2019).

[9] Angel Cañete, Jose-Miguel Horcas, and Lidia Fuentes. 2018. Mecanismos de Reconfiguración Eco-eficiente de Código en Aplicaciones Móviles Android. (2018).

[10] Hui Chen, Youhuizi Li, and Weisong Shi. 2019. pTopW : A Power Profiling Tool for Windows Platforms. (05 2019).

[11] Huangke Chen, Guipeng Liu, Shu Yin, Xiaocheng Liu, and Dishan Qiu. 2018. ERECT: Energy-efficient reactive scheduling for real-time tasks in heterogeneous virtualized clouds. *Journal of Computational Science* 28 (2018), 416 – 425. https://doi.org/10.1016/j.jocs.2017.03.017

[12] Z. Cheng, P. Li, J. Wang, and S. Guo. 2015. Just-in-Time Code Offloading for Wearable Computing. *IEEE Transactions on Emerging Topics in Computing* 3, 1 (March 2015), 74–83. https://doi.org/10.1109/TETC.2014.2387688

[13] Shaiful Chowdhury, Stephanie Borle, Stephen Romansky, and Abram Hindle. 2018. GreenScaler: training software energy models with automatic test generation. *Empirical Software Engineering* (20 Jul 2018). https://doi.org/10.1007/s10664-018-9640-7

[14] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic execution between mobile device and cloud. *EuroSys'11 - Proceedings of the EuroSys 2011 Conference*, 301–314. https://doi.org/10.1145/1966445.1966473

[15] Valerio Costamagna and Cong Zheng. 2016. ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime.. In *IMPS@ ESSoS*. 20–28.

[16] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*. ACM, New York, NY, USA, 49–62. https://doi.org/10.1145/1814433.1814441

[17] Krzysztof Czarnecki, Simon Helsen, and Ulrich 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 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spip.213

[18] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[19] T. Q. Dinh, J. Tang, Q. D. La, and T. Q. S. Quek. 2017. Offloading in Mobile Edge Computing: Task Allocation and Computational Frequency Scaling. *IEEE Transactions on Communications* 65, 8 (Aug 2017), 3571–3584. https://doi.org/10.1109/TCOMM.2017.2699660

[20] Ioana Giurgiu, Oriana Riva, Dejan Juric, Ivan Krivulev, and Gustavo Alonso. 2009. Calling the Cloud: Enabling Mobile Phones as Interfaces to Cloud Applications. In *Middleware 2009*, Jean M. Bacon and Brian F. Cooper (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 83–102.

[21] Youssef Hassoun, Roger Johnson, and Steve Counsell. 2004. Applications of dynamic proxies in distributed environments. *Software: Practice and Experience* 35, 1 (2004), 75–99. https://doi.org/10.1002/spe.629

[22] M. Jia, J. Cao, and L. Yang. 2014. Heuristic offloading of concurrent tasks for computation-intensive applications in mobile cloud computing. In *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 352–357. https://doi.org/10.1109/INFCOMW.2014.6849257

[23] Mads Darø Kristensen and Niels Olof Bouvin. 2010. Scheduling and development support in the Scavenger cyber foraging system. *Pervasive and Mobile Computing* 6, 6 (2010), 677 – 692. https://doi.org/10.1016/j.pmcj.2010.07.004 Special Issue PerCom 2010.

[24] E. A. Lee. 2008. Cyber Physical Systems: Design Challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 363–369. https://doi.org/10.1109/ISORC.2008.25

[25] Y. Lin, E. T. . Chu, Y. Lai, and T. Huang. 2015. Time-and-Energy-Aware Computation Offloading in Handheld Devices to Coprocessors and Clouds. *IEEE Systems Journal* 9, 2 (June 2015), 393–405. https://doi.org/10.1109/JSYST.2013.2289556

[26] Flavio Lombardi and Roberto Di Pietro. 2011. Secure virtualization for cloud computing. *Journal of Network and Computer Applications* 34, 4 (2011), 1113 – 1122. https://doi.org/10.1016/j.jnca.2010.06.008 Advanced Topics in Cloud Computing.

[27] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief. 2017. A Survey on Mobile Edge Computing: The Communication Perspective. *IEEE Communications Surveys Tutorials* 19, 4 (2017), 2322–2358. https://doi.org/10.1109/COMST.2017.2745201

[28] Y. Mao, J. Zhang, and K. B. Letaief. 2016. Dynamic Computation Offloading for Mobile-Edge Computing With Energy Harvesting Devices. *IEEE Journal on Selected Areas in Communications* 34, 12 (Dec 2016), 3590–3605. https://doi.org/10.1109/JSAC.2016.2611964

[29] Peter Mell, Tim Grance, et al. 2011. The NIST definition of cloud computing. (2011).

[30] Antti P. Miettinen and Jukka K. Nurminen. 2010. Energy Efficiency of Mobile Clients in Cloud Computing. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=1863103.1863107

[31] Claudia Pagliari, David Sloan, Peter Gregor, Frank Sullivan, Don Detmer, James P Kahan, Wija Oortwijn, and Steve MacGillivray. 2005. What Is eHealth (4): A Scoping Exercise to Map the Field. *J Med Internet Res* 7, 1 (31 Mar 2005), e9. https://doi.org/10.2196/jmir.7.1.e9

[32] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.

[33] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, and T. Taleb. 2018. Survey on Multi-Access Edge Computing for Internet of Things Realization. *IEEE Communications Surveys Tutorials* 20, 4 (2018), 2961–2991. https://doi.org/10.1109/COMST.2018.2849509

[34] Jan M Rabaey, Anantha P Chandrakasan, and Borivoje Nikolic. 2002. *Digital integrated circuits*. Vol. 2. Prentice hall Englewood Cliffs.

[35] Rydning Reinsel, Gantz. 2018. The Digitalization of the World: From Edge to Core. (2018).

[36] rovo89. [n.d.]. Xposed Framework. http://repo.xposed.info/. Online; accessed 13 May 2019.

[37] S. Sardellitti, G. Scutari, and S. Barbarossa. 2015. Joint Optimization of Radio and Computational Resources for Multicell Mobile-Edge Computing. *IEEE Transactions on Signal and Information Processing over Networks* 1, 2 (June 2015), 89–103. https://doi.org/10.1109/TSIPN.2015.2448520

[38] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (Oct 2016), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

[39] Madiha H. Syed and Eduardo B. Fernandez. 2015. The Software Container Pattern. In *Proceedings of the 22Nd Conference on Pattern Languages of Programs (PLoP '15)*. The Hillside Group, USA, Article 15, 7 pages. http://dl.acm.org/citation.cfm?id=3124497.3124515

[40] T. X. Tran and D. Pompili. 2019. Joint Task Offloading and Resource Allocation for Multi-Server Mobile-Edge Computing Networks. *IEEE Transactions on Vehicular Technology* 68, 1 (Jan 2019), 856–868. https://doi.org/10.1109/TVT.2018.2881191

[41] A. u. R. Khan, M. Othman, S. A. Madani, and S. U. Khan. 2014. A Survey of Mobile Cloud Computing Application Models. *IEEE Communications Surveys Tutorials* 16, 1 (2014), 393–413. https://doi.org/10.1109/SURV.2013.062613.00160

[42] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. 2014. Adaptive deployment and configuration for mobile augmented reality in the cloudlet. *J. Network and Computer Applications* 41 (2014), 206–216. https://doi.org/10.1016/j.jnca.2013.12.002

[43] Tim Verbelen, Tim Stevens, Pieter Simoens, Filip De Turck, and Bart Dhoedt. 2011. Dynamic deployment and quality adaptation for mobile augmented reality applications. *Journal of Systems and Software* 84, 11 (2011), 1871 – 1882. https://doi.org/10.1016/j.jss.2011.06.063 Mobile Applications: Status and Trends.

[44] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung. 2015. Dynamic service migration in mobile edge-clouds. In *2015 IFIP Networking Conference (IFIP Networking)*. 1–9. https://doi.org/10.1109/IFIPNetworking.2015.7145316

[45] Song Wu, Chao Mei, Hai Jin, and Duoqiang Wang. 2018. Android Unikernel: Gearing mobile code offloading towards edge computing. *Future Generation Computer Systems* 86 (2018), 694 – 703. https://doi.org/10.1016/j.future.2018.04.069

[46] K. Zhang, Y. Mao, S. Leng, Q. Zhao, L. Li, X. Peng, L. Pan, S. Maharjan, and Y. Zhang. 2016. Energy-Efficient Offloading for Mobile Edge Computing in 5G Heterogeneous Networks. *IEEE Access* 4 (2016), 5896–5907. https://doi.org/10.1109/ACCESS.2016.2597169

[47] W. Zhang, Y. Wen, and D. O. Wu. 2013. Energy-efficient scheduling policy for collaborative execution in mobile cloud computing. In *2013 Proceedings IEEE INFOCOM*. 190–194. https://doi.org/10.1109/INFCOM.2013.6566761

[48] W. Zhang, Y. Wen, and D. O. Wu. 2015. Collaborative Task Execution in Mobile Cloud Computing Under a Stochastic Wireless Channel. *IEEE Transactions on Wireless Communications* 14, 1 (Jan 2015), 81–93. https://doi.org/10.1109/TWC.2014.2331051

[49] Tianchu Zhao, Sheng Zhou, Xueying Guo, Yun Zhao, and Zhisheng Niu. 2015. A Cooperative Scheduling Scheme of Local Cloud and Internet Cloud for Delay-Aware Mobile Cloud Computing. (11 2015). https://doi.org/10.1109/GLOCOMW.2015.7414063

[50] X. Zhu, L. T. Yang, H. Chen, J. Wang, S. Yin, and X. Liu. 2014. Real-Time Tasks Oriented Energy-Aware Scheduling in Virtualized Clouds. *IEEE Transactions on Cloud Computing* 2, 2 (April 2014), 168–180. https://doi.org/10.1109/TCC.2014.2310452

Facilitating the Development of Software Product Lines in Small and Medium-Sized Enterprises

Nicolas Hlad
LIRMM
Montpellier, France
hlad@lirmm.fr

ABSTRACT

Software Product Lines (SPLs) are Software Engineering methodologies that manage the development and evolution of families of product variants. They aim at handling the commonality and variability of these products. SPLs reduce the development cost, time-to-market, and increase overall quality of the product variants.

But Small and Medium-sized Enterprises (SMEs) can find the development of an SPL to be expensive and challenging, especially the process regarding the domain engineering. They can be forced to hire or train dedicated SPL-experts to work on the SPL development. This extra cost can be a significant obstacle toward the adoption of this technology by these enterprises.

In this paper we present our work on an new approach to reduce and facilitate the adoption of the SPL techniques in SMEs. The goal of this approach is to automatically and incrementally build SPLs. This is based on an original combination of existing extractive and reactive approaches.

We advocate the need for new solutions to facilitate the adoption of SPL technology by SMEs. We discuss a global solution based on an original combination of the existing extractive and reactive approaches. Our solution is a new approach that automatically and incrementally build SPLs. We present our global solution under the form of five research questions, of which we discuss the motivations and the methodologies. We also present early results of the first three questions, while the remaining two are the matter of future work.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

Software Product Line, Feature, Asset, Reactive, Extraction

ACM Reference Format:
Nicolas Hlad. 2019. Facilitating the Development of Software Product Lines in Small and Medium-Sized Enterprises. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3307630.3342703

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

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342703

1 INTRODUCTION

Software Product Lines (SPLs) manage the development of families of product variants, by handling their commonality and variability. SPLs enable lower costs, shorter time-to-market and higher quality for the SPL products [20].

Each SPL is defined by its architecture, which describes how common and variable parts interact with each other. Similarly to any other programming projects, the SPL architecture is implemented using source code files. Inside this architecture, variability is commonly represented using features and variation points.

A feature is *a distinguishing characteristic of a product. It can be functional (like an action) or qualitative (like a security level)* [8, 12]. In the rest of this paper, we will consider that a feature is implement in the SPL architecture using one or more assets. An asset describes source code, test files, design model, etc. However in our works, we only look at assets of code. Because each feature is distinguishable from the others, it means that inside one SPL, two different features are made of two different sets of assets. And finally, a feature is rarely implemented in one specific place inside the architecture, thus its assets can be distributed all over the SPL architecture.

Variation points represent these specific places in the SPL architecture where one product variant can differ from the others ([20] p.62). To generate a product variants from the SPL, variation points need to be resolved using a configuration. The configuration indicates which feature should be include or exclude from the variants. This process is also refer as the derivation.

The SPL development cycle is divided in three process: scoping, domain engineering and application engineering [5, 20].

Scoping is the process of gathering information surrounding the SPL domain. For example, developing an SPL on medical application will require gaining knowledge about the medical domain. This can be about the type of data, domain processes, specific features, etc. The output of this process is a deep knowledge of the domain, and a clear view of what the SPL will address and what it will left aside.

Domain engineering is the process where knowledge provided by the scoping is used to implement the SPL architecture. Inside this process, the commonality and variability are defined and turn into assets.

Finally, *application engineering* is the process where products are built by reusing the assets and by exploiting the variability of the SPL. This process can output new requirements that are based on the product's deployment. These requirements will be implement in a new iteration of the domain engineering.

Of these three processes, domain engineering is the Achilles' heel of SPLs development. It is expensive in resources and time. If not handled properly, it can result in an SPL failure ([20] p.9-10). We distinguish three types of approaches that are commonly used

to build an SPL. Each of these approaches have a direct impact on how domain engineering is achieved:

Proactive approaches are essentially used by software editor companies. These approaches take as an input a deep and robust scoping of the domain, in order to anticipate all the *requirements* ([8] p.62). Those requirements are all implemented as assets at once and is done by SPL-experts[1].

Extractive approaches are an automated process that take as an input a set of existing product variants. The SPL is built by extracting the features from theses variants. These approaches are used by companies which already have product variants built from ad-hoc methods [10]. Features are retrieved using feature location techniques [1, 21].

Finally, *reactive approaches* are based on an incremental process. They take as inputs an SPL_i (SPL version i) and a set of new requirements related to a new product. This process outputs SPL_{i+1}, which is able to generate the new product. These approaches allow companies to significantly reduce the up-front investment that comes with proactive approaches [6]. With the reactive approaches, the scoping can be limited to an overview of the domain, only requiring the implementation of a few essential assets.

With the reactive approaches, building the assets is still the work of the SPL-experts. The cost of the domain engineering process is the same as in proactive approaches, but only spreads over time. Furthermore, reactive approaches require SPL-experts to directly modify the assets, which is more complicated than to work on the development of a single product [13, 18].

Numerous case studies show the advantages of SPL for large companies, and specially for software editor companies ([20] chapter 21). However, trying to export these approaches to Small and Medium-sized Enterprises (SMEs) appears to be more complicated. Jansen and al. [11] even suggest that SPLs are not the answer for SMEs. In fact they should instead focus on opportunistic reuse of existing third-party software-components. Clement and al. [7] have established a road map on how to successfully apply SPLs in a company. Their paper declares that companies that are not fully devoted to follow the SPLs principles, will fail in their adoption. This means that companies need to restructure their development teams, management processes and the development process as an all. Small-sized enterprises, like a startup, may be able to restructure their teams faster than the others. But as Jansen and al [11] argues, startups are driven by getting a return of investment as fast as they can, and SPLs simply take too much of their resources. Njima and al [19] have observed similar behaviors in their study on two startups trying to adopt SPL development.

Medium-sized enterprises should be an interesting place to apply SPLs: they have more resources than small-sized enterprises and have more skilled developers. But their development cycle[2] is also regular as clockwork, meaning that they are unwilling to risk changing it if there is any riskof losing profits.

In this paper, we want to focus on the problem of SPL adoption in SMEs. In section 2, we formulate the research questions we want to address during the first half of this PhD. In section 3 we present the state of our research. The section 4 discusses our plans for future works. Finally, we conclude in section 5.

[1]Developers or Engineers specialized in SPL development

[2]By *development cycle*, we refer at the production process the company use to develop their products.

2 RESEARCH QUESTIONS

In the introduction, we have seen that initiating SPLs in SMEs is challenging. On one hand, it requires a lot of efforts to restructure the development cycle. On the other hand, it does not guaranty early return of investments for these companies.

Hypothesis

We know that SPLs can have a real advantage at being used in the development cycle, but how can we convince SMEs to apply them? If SPLs bring more short term challenges than long term benefits, SMEs will have problems perceiving the values this solution. We hypothesize:

> **Hypothesis**: We can define an approach to build software product lines that does little to no impact on the development cycle uses by a company

Research Questions

To validate our hypothesis, we formulate the following Research Questions (RQs):

- **RQ1**: What are the criterion that compromise the adoption of SPLs in the SMEs product development? (Study-1)
- **RQ2**: What aspects of the existing approaches should be applied in the SMEs context? (Study-2)
- **RQ3**: What would be an ideal new approach for SMEs? (Study-3)
- **RQ4**: Can we implement this approach into a tool? (Study-4)
- **RQ5**: How can we retrieve the features from this new approach? (Study-5)

With these RQ, we focus on the definition of a new approach that will help SMEs deploying SPLs at the relatively low cost. This way, we think that SMEs would have little to no challenges to face regarding the use of SPL in their development cycle.

Expected Contributions

By answering these RQs, we expect our work to produce the following contributions:

- A set of criterion that causes the adoption problem in SMEs.
- A new approach to develop SPLs that can be apply in any company, regardless of their actual development process.
- Tools that implement this new approach.

3 RESEARCH METHODOLOGY AND PRELIMINARY RESULTS

This PhD is supported by a medium-sized partner company. This company offers IT services to multiple clients in large areas of business domain. We will refer to this company as our *partner company* and most of our experiments will take place there.

In this section, we detail how we address our research questions. For each question, we present the motivation, the research methodology, and the preliminary results.

3.1 Study-1 : The Adoption of SPLs by SMEs

3.1.1 Motivation. We focus first on the challenge surrounding the adoption of SPLs by SMEs. We suppose that it lies in the difficulties

of implementing the domain engineering. Methods like PULSE [13] have demonstrated that a successful scoping will certainly improve the domain engineering. However, this does not address the cost of developing the domain engineering. The difficulties lie in the implementation of meaningful assets that will be reused, while minimizing the cost of their implementation. Often, SMEs adopt reactive approaches, but they are not able to anticipate the assets they will need. So they rely on new requirements as opportunities to implement important assets.

3.1.2 Methodology. Study-1 is about searching for difficulties in SPL development. We address RQ1 by observing our partner company building an SPL. A training course on SPL engineering is proposed to a team of two experienced developers and their manager. Then, they are ask to implement a product line based on a real life project. We meet the team once every two weeks for 3 months to get their feedback. These meetings help us identify the SPL aspects that were not clear to them. The main problem highlighted by the developers was the difficulty concerning the actual implementation of the SPL.

3.1.3 Threats to validity. This study as been conducted with limited times and resources and thus cannot be considered as a proper empirical study. We did not have access to a large enough sample of SMEs or developers to collect a sufficient amount of data. The analyses and results of this study should not be generalized to all SMEs. We plan to later remake this study by applying proper experimental protocols from empirical methodologies. Therefore, our results can only be considered as *early results*.

3.1.4 Early results. We observe that developers have trouble switching from single products development to an SPL-oriented development. For example, features were implemented with one product in mind, sometimes resulting in the re-development of the same feature inside two product variants. We also notice that the lack of proper testing and static analysis tools is a serious issue, as they have to generate each product variant to test them. To quote the team, this results in "*a waste of time*".

An another concern is about the time of development impacting the overall cost. SMEs, specially IT service companies, charge their works by the hour. As the time to implement a feature increases, so does the cost. This was expected and our partner knew about it, but they still reported this concern.

Furthermore, we observe reluctance from the team, which is focusing on product delivery rather than the implementation of reusable assets. When asked about this issue, the team answered that they "*do not see what can be reuse*". We believe that this is due to lack of time during the training course when we explain the necessity of the domain engineering process.

Finally, we also report that the team did not pursued in the implementation after 1 months of attempts. They left it aside and preferred focusing on a single product development.

This experiment shows that SPLs are not easy to implement, they require time and training. Teams need to be trained at the SPL-oriented development. We argue that experienced developers will not change their habits and only new developers will be able to make the jump from a single product development cycle to an SPL development cycle. New developers may not have the same reluctance but they will have to be trained very early with SPL development. However, SMEs may not have the resources to train them.

We summarize our observations in two statements : (a) *Experienced developers will not adopt SPLs if it means changing their development habits*; (b) *SMEs won't adopt SPLs if the up-front investment is too high*. Statement (a) can be explained by the lack of tools for testing and debugging, while statement (b) is explained by the investment in the training of the teams, and the cost of SPL-oriented development.

3.2 Study-2: Searching for useful aspects among the existing approaches

3.2.1 Motivation and Methodology. We have observed that SMEs have concerns with initiating SPLs because of the initial cost and the difficulties faced during the SPL-oriented development. Based on Study-1, we search for aspects inside the existing approaches (proactive, extractive, and reactive approaches) that could resolve statements (a) and (b).

3.2.2 Results. With (a), we learn that experienced developers will prefer an approach that does not change their habits. We also learn that there is a lack of professional tools to manage the development of SPLs (like debugging and testing features). In extractive approaches, *products are independently developed from any SPL Engineering methodology*. Those products are developed using existing integrated development environments (IDEs). This aspect of extractive approaches allow developers to work on a single product development cycle, thus resolving (a).

And with (b), we learn that the up-front investment of building the SPL is an obstacle for SMEs. Among those existing approaches, only the reactive ones allow for a *significant reduction in the up-front investment*, but at the cost of foreseeing future requirements. Furthermore, reactive approaches focus on a *rapid return of investment*, which is appreciated by SMEs. Theses two aspects should resolve (b).

By combining the aspects of extractive and reactive approaches, we can define a new hybrid approach that allows developers to work on a single product development, while building the SPL from a reactive process. The combination of these two approaches offer solutions to the statements (a) and (b).

3.3 Study-3: Defining a new approach

3.3.1 Motivation. We want the approach to have two contributions: First i) we stated in Study-2 that by combining extractive and reactive approaches, the up-front cost should be significantly lower than proactive approaches (also stated here [6]). If the SPL production adds no cost to the current development process, maybe it will turn as an argument in favor of the SPL adoption in some SMEs. Second ii) we want to create an approach that as little to no impact on the current development cycle of a company. In order to go further regarding (a), we consider that any intrusive approach, that could disturb an already establish development process, will participate in favor of an adoption failure. We are searching for an approach that can be graft to any development process. To assure ii), we consider having no prior knowledge on the product configurations or on existing features.

To help us define our approach, we start by looking at the representation of an SPL. We made a UML our SPL representation in the figure 1. This figure represents all the entities presented in this approach and how they relate to each other.

Features are an abstract concept and are loosely define in the code, while assets are the actual implementation of the features in the code. We affirm that in one single SPL, two distinct features are implemented by two different sets of assets. Therefore we can defined an approach that extract these assets and distinguishes common from optional assets by looking at the differences inside the code of two products.

In our approach, we want to remove the need of SPL-experts because the development cycle is made on a single product at a time. Each product is built from a reactive approach angle, where new requirements come as opportunities to improve the SPL. Thus, any developer can implement a product variants and contributes to the SPL without directly working on the SPL architecture.

Once a product is delivered, the approach takes its extractive angle and extracts the assets from the product. These assets are used to automatically build the SPL. But in order to guaranty a relation between each new product, we ask developers to generate a *product-base* from the SPL each time a new requirement appears. The product-base is an incomplete product (regarding the requirements), that is used as a foundation to develop the new product. Thus, each products is made from reused assets and (if needed) from newly implemented assets.

3.3.2 Related Work. A large number of contributions on how to extract SPLs have been proposed over the last decades. We look at studies that extract SPLs while requiring little to no knowledge about it and have a limited number of product variants at their disposal.

Valente and al. [22] propose a semi-automated approach, where developers identify *seeds* of an already known feature (equivalent to our assets). The seeds are given to an algorithm to located the complete feature implementation related to the seeds. It also extracts parts of the feature-model, but requires inputs from developers when a conflict is detected. This approach focus on recovering the traceability between a feature and its implementation, using at least one product. According to the authors, they can efficiently recover common features but have problems with the optional features. Furthermore, this study does not address the evolution aspect of an SPL.

Similarly, Linsbauer and al [17] propose the traceability extraction and the recovering of the feature-model using an existing set of product variants. In their approach, the configuration of each product is known, as well as a list of the definitive feature.

We differ from these studies by having no prior knowledge of the existing features, or product configuration. Furthermore we build the SPL inside a reactive approach, which means that our SPL can radically evolve over time. Both studies apply their techniques on existing product variants, and on what it seems to be a final product line.

3.3.3 Methodology. We use the observations and feedback collected from our investigation on Study-1, and the aspects find in Study-2, to define our new approach. To test our approach, we plan to use empirical testing with the help of a prototype that implements this approach (Study-4). We also use our partner company to support the validity of our approach. We make the hypothesis that if our approach is successfully adopted by our partner, then it could be exported to other SMEs. But it would require a large population of SMEs that experiments with our approach to fully validate it.

3.3.4 Early Results. We have produce an overview of this approach, represented in figure 2.

In our approach, we look at assets as source codes of the SPL. Common assets are always present in each product of the line, while the presence of optional assets is variable and depends on the product configuration.

Initialization with Two Products: We start with at least two products (like show in figure 2 part 1), build from ad-hoc method (similar to the *Clone&Own* technique [10]). Thus, these two products share common and distinct assets. They share similar files hierarchy and software architectures, which make them comparable. This part of the process (part **1** in igure fg:generalProcess) is only done once, at the beginning.

For example, listing 1 shows the Java class *Number* in two products P1 and P2.

```
1  //P1
2  class Number {
3    int n;
4    void decrease() { n=n-1; }
5  }
6  ------------------------------
7  //P2
8  class Number {
9    int n;
10   void increase() { n=n+1; }
11 }
```

Listing 1: Product P1 and P2

Extracting Assets: We consider that each feature is implemented with an unique set of assets. Based on our hypothesis, there is a bijective relation between features and code assets. We identify the features by extracting their assets. Extracting the assets is possible by comparing the code of each product and search for differences.

We start by extracting all the assets from the two products. The types of assets will depend on the language used. So we define different type of granularity for the different type of assets. For example, in Java, we extract the assets the size of : a package declaration, an import declaration, a class, an attribute, and a method. Therefore we need to specify a catalog of the assets granularity for each programming language.

In order to differentiate common assets from optional assets, we compare their codes by analyzing the Abstract Syntax Tree (AST) of the two products. Therefore, code format can be different between two files and the comparison will only detect differences in the AST structure. For example, if any difference is highlighted in a method fields, we set all this method as an optional assets.

In our example, listing 2, we consider our assets granularity to be the size of classes, methods and attributes. We use the AST comparison on $P1$ and $P2$ to find the methods *decrease* and *increase* as two optional assets. The rest of the assets are common.

```
1  //P1
```

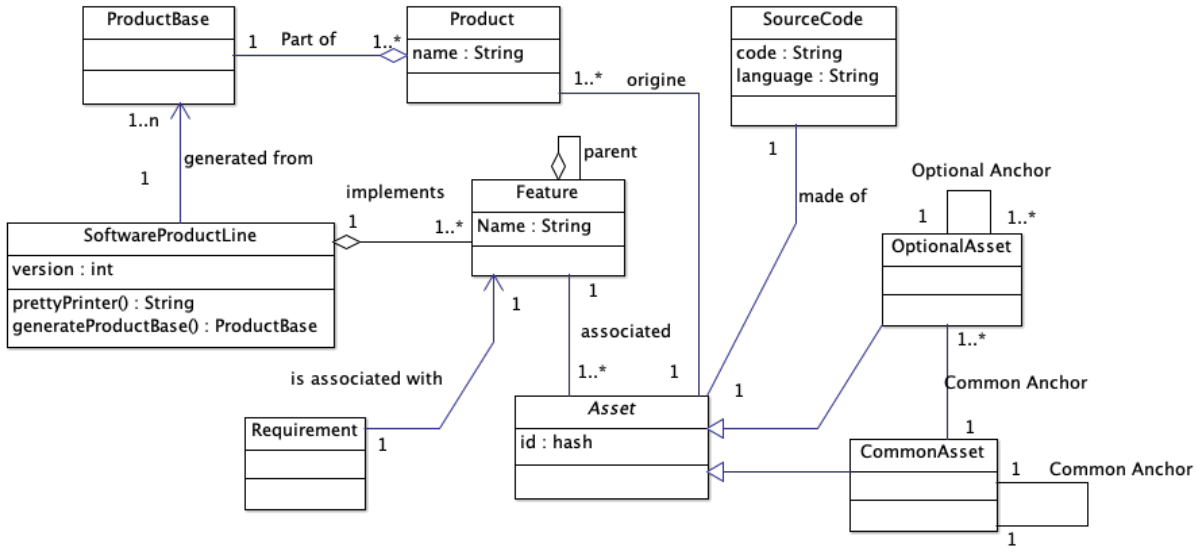


Figure 1: A class diagram of our representation of SPLs

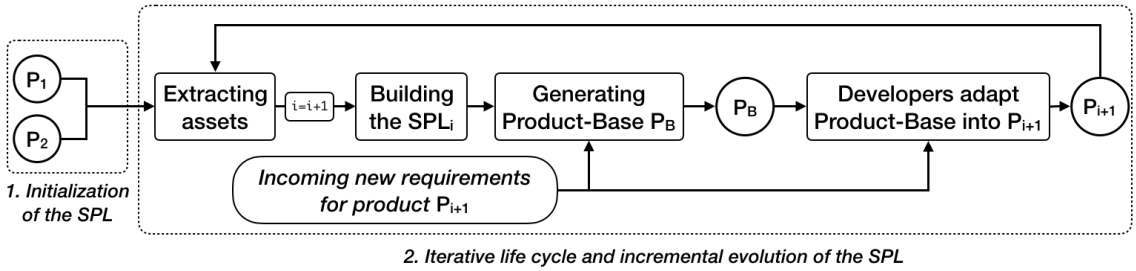


Figure 2: Life cycle of our approach. part 1 describe the initialization and is also done once at the beginning of the approach. Part 2 is the iterative process that enable the incremental evolution of the SPL.

```
class Number {
  int n;
    void decrease() { n=n-1; }
}
---------------------------------
//P2
class Number {
  int n;
    void increase() { n=n+1; }
}
```

Listing 2: Result of the AST-based differences on products P1 and P2.

Building the SPL: In our approach, an SPL architecture can be seen as a collection of linked common and optional assets. We take the output of *Extracting Assets* and build the SPL by comparing our sets of assets (one set for each product variant). We obtain:

$P1 : \{\varnothing \leftarrow$ **class Number** $\leftarrow$ **int n** $\leftarrow$ *decrease*$\}$

$P2 : \{\varnothing \leftarrow$ **class Number** $\leftarrow$ **int n** $\leftarrow$ *increase*$\}$

In this example, we have two optional assets : the methods *decrease* and *increase* (in *italic*), and two common assets : the classes *class Number* and its attributes *int n* (in **bold**).

Because we have extracted our assets using the AST and we have extracted them with a specific granularity, we can replace them into the SPL architecture using the language grammar. For example in Java, a method will always be inside a class, a class precede by the imports, etc.

And we also link our assets together using *Anchor*, which give them a relative position (this relation is symbolized by $\leftarrow$ [3]). We use two types of anchor: the *common anchor* and *optional anchors*

An common anchor is the previous common asset, right before the asset we are looking at. Here, *decrease* and *increase* have for anchor the attribute *int n* and thus will always be positioned after *int n*. The first common asset (here the *class Number*) has an empty anchor. Optional assets can share the same anchor (like *decrease* and *increase*).

Optional anchors are used to position an optional asset depending on other optional assets before it. We create these anchors when

[3] $anchor \leftarrow asset$

we extract two following optional assets inside one product. Optional assets can have multiple optional anchors because we can find different sets of assets at each new product encounter. We only use optional anchors on assets that requires one. For example, in Java, import declarations need to respect a specific order, while it is not the case for methods or attributes.

Assets also have an unique identifier. Therefore, we can compare assets with this identifier and remove duplicates (here *class Number* and *int n* appear in both the set of $P1$ and $P2$). Assets that have turned from common to optional are kept as optional in the final set. The output of this process is a single set of unique assets for the SPL:

$SPL_1 : \{\varnothing \leftarrow \textbf{class Number} \leftarrow \textbf{int n} \leftarrow \{decrease, increase\}\}$

One thing we do not address yet in our approach is the retrieval of the feature and the feature-model. As part of our work in progress we will focus on this concern in 3.5 with Study-5. For now, we assume that Features and the feature-model can be retrieve during this process.

(optional) Pretty Printing the SPL architecture: Developers may want to manipulate the SPL architecture directly. Our approach allows the SPL architecture to be generated inside source code files.

To generate product from the SPL, we need to define where are the variability points and what are their representation. In our case, variation points appear each time we find a new optional asset. To represent variation points, different methods and tools [4, 14, 16] exist. Those tools and methods are based on two approaches which are the *compositional* and the *annotative* [15]. The compositional approach physically isolates assets in specific folder with source files inside, each folder matches a feature. When the SPL is derived, codes are *composed* to build the product. The annotative approach resembles C-preprocessing macros. Assets are logically isolated by annotations inside the SPL architecture. annotations have names that match a specific feature. During the derivation, desired annotations (with their associated code) are kept in the architecture while the others are removed. In our approach, we choose to use the annotative approach as it is more simple to use for developers and only requires to put annotations in the code surrounding the assets [2, 15]. Listing 3 shows the SPL architecture with the assets $A1$ and $A2$ extracted from $P1$ and $P2$. These optional assets are isolated from the common assets using opening and closing annotations (here $if[Ax]$ and $end[Ax]$). Listing 4 shows the product-base of P3 that contains assets $A1$ and $A2$, as asked by the requirements.

```
1  //SPL code at iteration 1
2  class Number {
3    int n;
4    /*if[A1]*/
5    void decrease() { n=n-1; }
6    /*end[A1]*/
7    /*if[A2]*/
8    void increase() { n=n+1; }
9    /*end[A2]*/
10 }
```

Listing 3: Source code of the SPL from previous example in listing 1 and 2

Incoming New Requirements: According to the principals of reactive approach, new requirements trigger an evolution of the SPL. We use them to influence the generation of new products. The goal is to generate a product-base from the SPL that will be as close as possible to the required new product. Using the SPL to generate the product-base guarantees that our final product will be part of the product family. It also guarantees the reusability of our SPL assets. Having this product-base allows developers to work on a single product variant, rather than directly on the SPL to generate the new product.

In our SPL example, the new fictional requirements are to make *a Product P3 where the class Number can increment, decrease and add two Numbers.*

Generating the Product-Base: We represent our SPL architecture as a set of optional and common assets, linked together by anchors. Therefore our variation points are logical rather than physical : any optional assets is associated with a variation point.

Ultimately, assets are also associated with features, and the configuration space (space of all the possible configuration offers by the SPL) is represented by a feature-model [3].

To resolve the variability, and thus derive the SPL into a product variant, developers need to specify which features should remained in the product, using guidelines from the feature-model. This action is performed using a configuration file. The desired assets are *pretty-printed* into files to create the architecture of the product-base. The pretty-printer used here follow the same principles as the one used to pretty print the SPL architecture.

In our approach, the product-base guaranties a direct reusability of assets. It also serves to prevent the *out-of-scoop* effect of the SPL. The effect happen when developers try to force a totally unrelated product inside the SPL. By using the product-base, we want to force new products to be related with the current SPL.

In our example the generated product-base is PB_3, which contains the some of the assets matching the requirements of our example.

```
1  //product-base of P3, generated from the SPL
2  class Number {
3    int n;
4    void decrease() { n=n-1; }
5    void increase() { n=n+1; }
6  }
```

Listing 4: code of the generated product-base $P3$ including features $A1$ and $A2$

Developers adapt the product-base into the final product: The product-base is likely to miss some of the new product requirements. Developers simply need to adapt this product and fulfill these requirements. In listing 5 we see the final product $P3$, with its the new method *add*.

```
1  //P3 final code.
2  class Number {
3    int n;
4    void decrease() { n=n-1; }
5    void increase() { n=n+1; }
6    int add(Number nb){return nb.n+n;}
7  }
```

Listing 5: Product $P3$ after completion (here the method add() is implemented)

Incremental Evolution of The SPL: Now that we have a new final product, we can incrementally evolve the SPL. After extracting assets of the new product, we compare it with a *pretty printed SPL*

that only contains common assets. This is because common cannot be added[4], therefore there is no need to recompare optional assets between them. The new product can carry two pieces of information : i) we can find that a common asset is in fact optional because it does not appear in the new product. And ii), that the new product implements a new optional asset.

Each comparison of two products gives us a set of common and optional assets (as explained earlier). By merging each set into one, we build the new version of the SPL : SPL_2 : $\{\varnothing \leftarrow$ **class Number** $\leftarrow$ **int n** $\leftarrow \{decrease, increase, add\}\}$ To represent this new version of the SPL in our example we can pretty-print its architecture, which is showed in listing 6.

```
1  //SPL code at iteration 2
2  class Number {
3    int n;
4    /*if[A1]*/
5    void decrease() { n=n-1; }
6    /*end[A1]*/
7    /*if[A2]*/
8    void increase() { n=n+1; }
9    /*end[A2]*/
10   /*if[A3]*/
11   int add(Number nb){return nb.n+n;}
12   /*end[A3]*/
13 }
```

Listing 6: Annotative representation of the SPL architecture

3.3.5 Threats of Validity. Our approach wants to start with no prior knowledge of the SPL. We have hypotheses that it would be possible to recover the entire product line without asking any input from developers. But this hypothesis may shows it limits in several process: i) we have not yet find a way to recover the features from the assets and Study-5 has been made for this purpose. ii) Naming the feature will ultimately require the validation from developers. iii) We do not yet know what happen if an asset is replaced by developers. Because developers cannot delete existing optional assets, this may result in some *zombie assets* inside the SPL architecture. A zombie asset will influence the rest of the architecture even if it is never reused inside any product-base.

Finally we need more experimentation on our approach and this section only describes its core concepts.

3.4 Study-4 : Implementing our approach

3.4.1 Motivation. Study 1 and 2 show us that the cost of developing an SPL was an important obstacle for the SMEs. Since then, we have focus our effort in reducing that cost. We think that having an automated approach, such as the one proposed in Study-3, will help solve the cost issue. In this study, we look at how to implement our approach into a prototype.

3.4.2 Methodology. We choose to start from existing technologies and to use academic tools. We want our prototype to use open-source software components, this way we can easily make evolution on these tools if needed. We first implement the UML representation of our SPLs (figure 1) in a software and then we added the routine of the approach.

[4]It is impossible to find a new common asset inside a new product variant, because this asset is not common in the rest of the products line.

3.4.3 Results. We have implemented our prototype in a Java. We use Gumtree [9] to do the diff aspect of our approach. Gumtree is a diff tool based on the AST of the code. It is an academic tools and can be considered as state of the art in this domain. Using Gumtree, we extract all our assets, as described in Study-3.

This prototype compares product variants two-by-two, but it only work with product made in Java. The extracted assets from each product are used to assemble the SPL. Then, we generate the product-base from the SPL and build the next product, using the product-base as a foundation.

We use Munge [23] to generate the annotative representation of our SPL. Munge is a pre-processor code parser for Java code. It looks for annotations across the code. For example, in listing 2 and 6 we use the Munge's annotations ($if[Ax]end[Ax]$). annotations are associated with assets, and using Munge, we can keep or remove the assets associated to a specific tag. However, the prototype is not finished and more work is required to achieve a version that would support the experimentation.

3.5 Study-5: Recovering features from assets

3.5.1 Motivation. Our approach (described section in Study-3) omits two essential aspects of a SPL: the *feature location* and the *feature dependency*. Feature location is the process of searching the actual implementation of a feature. While the feature dependency is the process of finding the relations between the features.

We often represented the dependency using a *Feature Model* (FM) [3]. FM are a hierarchical representation of the dependency of each feature. They also give the state of the features, such as if a feature is mandatory, optional, under some condition, etc. Building the corresponding FM of a constantly evolving SPL is tricky and requires analyzing the interaction of the features between them.

3.5.2 Methodology. We hypothesize that starting from the assets, it will be possible to reduce the space of research for the feature location in the code. For example, techniques such as FCA [1] can be apply on the set of assets rather than the entire code. We plan on clustering the assets into groups that could match the features. To retrieve the Feature Model, we plan on using an approach where we are using the dependency find during the assets extraction. We will probably have to generate all the possible product variants from the SPL and study the assets dependency on each of them.

4 WORK PLAN

This PhD has started in October 2018 and will last until October 2021. On May 2019, we have completed Study-1 to 3, and we are now focusing on finishing Study-4. Figure 3 represents our planning for the next 12 months. Study-4 is divided in the implementation and the experimentation, which would end around November 2019. The rest is dedicated to Study-5, which is divided into the Feature Location and Feature Model problems. Study-5 is a uncertain task and may takes a lot more time to achieve.

5 CONCLUSION

In this paper we have presented our work on improving the adoption of software product lines for the small and medium-sized enterprises. To some extends, our work aims at being adapted toward

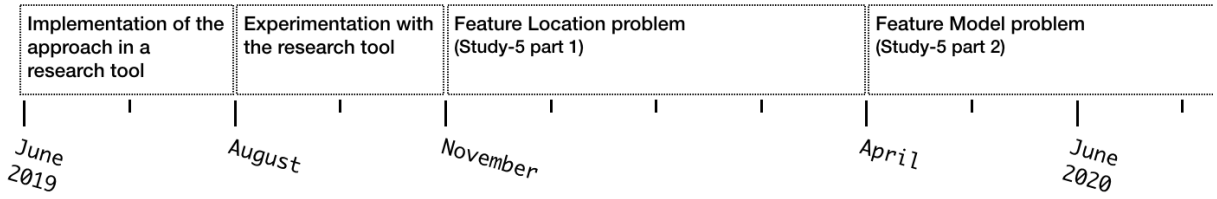


Figure 3: Time-line representation of our 12 months work plan.

any company without disturbing their existing development cycle. We have developed an original hybrid approach based on the reactive and extractive ones, to address two observed problems : a) *Experienced developers won't adopt SPLs if it means changing their development habits*, and b) *SMEs won't adopt SPLs if up-front investment is too high.*

By merging the right aspects of the reactive and extractive approaches, we have been able to develop an approach that works to address a) and b). It does not require developers to change their habits and do not require SMEs to spend too much resources in an up-front investment.

However our approach has yet figured a way to recover the features and feature-model of the product line. Assets alone are unpractical to manage SPLs. In our future work, we will focus on feature location, based on these assets, in order to retrieve their associated features and the feature model of the product line.

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 - 13th International Conference on Software Reuse, ICSR 2013, Pisa, Italy, June 18-20. Proceedings*. 302–307. https://doi.org/10.1007/978-3-642-38977-1_22

[2] Fellipe Araújo Aleixo, Marília Aranha Freire, Daniel Alencar da Costa, Edmilson Campos Neto, and Uirá Kulesza. 2012. A Comparative Study of Compositional and Annotative Modelling Approaches for Software Process Lines. In *26th Brazilian Symposium on Software Engineering, SBES 2012, Natal, Brazil, September 23-28, 2012*. 51–60. https://doi.org/10.1109/SBES.2012.11

[3] Don S. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*. 7–20. https://doi.org/10.1007/11554844_3

[4] Danilo Beuche. 2007. Modeling and Building Software Product Lines with pure: : variants. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings. Second Volume (Workshops)*. 143–144.

[5] Jan Bosch. 2000. *Design and use of software architectures - adopting and evolving a product-line approach*. Addison-Wesley.

[6] Ross Buhrdorf, Dale Churchett, and Charles W. Krueger. 2003. Salion's Experience with a Reactive Software Product Line Approach. In *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers*. 317–322. https://doi.org/10.1007/978-3-540-24667-1_24

[7] Paul C. Clements, Lawrence G. Jones, John D. McGregor, and Linda M. Northrop. 2006. Getting There from Here: A Roadmap for Software Product Line Adoption. *Commun. ACM* 49, 12 (Dec. 2006), 33–36. https://doi.org/10.1145/1183236.1183261

[8] Institute Electrical and Electronics Engineers. 1990. Glossary of Software Engineering Terminology, IEEE Standard 610.12. (09 1990). https://doi.org/10.1109/IEEESTD.1990.101064

[9] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982

[10] Eddy Ghabach. 2018. *Supporting Clone-and-Own in software product line. (Prise en charge du « copie et appropriation » dans les lignes de produits logiciels)*. Ph.D. Dissertation. University of Côte d'Azur, Nice, France. https://tel.archives-ouvertes.fr/tel-01931217

[11] Slinger Jansen, Sjaak Brinkkemper, Ivo Hunink, and Cetin Demir. 2008. Pragmatic and Opportunistic Reuse in Innovative Start-up Companies. *IEEE Software* 25, 6 (2008), 42–49. https://doi.org/10.1109/MS.2008.155

[12] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. (1990).

[13] Peter Knauber, Dirk Muthig, Klaus Schmid, and Tanya Widen. 2000. Applying Product Line Concepts in Small and Medium-Sized Companies. *IEEE Software* 17, 5 (2000), 88–95. https://doi.org/10.1109/52.877873

[14] Charles W. Krueger and Paul C. Clements. 2014. Systems and software product line engineering with gears from BigLever software. In *18th International Software Product Lines Conference - Companion Volume for Workshop, Tools and Demo papers, SPLC '14, Florence, Italy, September 15-19, 2014*. 121–125. https://doi.org/10.1145/2647908.2655976

[15] Christian KÃďstner and Sven Apel. 2008. Integrating Compositional and Annotative Approaches for Product Line Engineering.

[16] 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, ETX 2005, San Diego, California, USA, October 16-17, 2005*. 55–59. https://doi.org/10.1145/1117696.1117708

[17] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2018. Variability extraction and modeling for product variants. In *Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018*. 250. https://doi.org/10.1145/3233027.3236396

[18] John D. McGregor, Linda M. Northrop, Salah Jarrad, and Klaus Pohl. 2002. Guest Editors' Introduction: Initiating Software Product Lines. *IEEE Software* 19, 4 (2002), 24–27. https://doi.org/10.1109/MS.2002.1020282

[19] Mercy Njima and Serge Demeyer. 2019. An Exploratory Study on Migrating Single-Products towards Product Lines in Startup Contexts. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS 2019, Leuven, Belgium, February 06-08, 2019*. 10:1–10:6. https://doi.org/10.1145/3302333.3302347

[20] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer. https://doi.org/10.1007/3-540-28901-1

[21] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering, Product Lines, Languages, and Conceptual Models*. 29–58. https://doi.org/10.1007/978-3-642-36654-3_2

[22] Marco Tulio Valente, Virgilio Borges, and Leonardo Teixeira Passos. 2012. A Semi-Automatic Approach for Extracting Software Product Lines. *IEEE Trans. Software Eng.* 38, 4 (2012), 737–754. https://doi.org/10.1109/TSE.2011.57

[23] Jesse Wilson and Tom Ball. 2009. Preprocessing .java with Munge. https:\\publicobject.com/2009/02/preprocessing-java-with-munge.html. Accessed: 2018-11-11.