Reasoning About Edits to Feature Models (ICSE'09)

SPLC'23 MIP Award Talk | Thomas Thüm, Don Batory, Christian Kästner | August 31, 2023

universität ulm

TEXAS
The University of Texas at Austin

Carnegie Mellon University

Before We Start

I recently read that ...

it is not healthy to sit down all day

Before We Start

I recently read that . . .

it is not healthy to sit down all day

I propose that you . . .

stand up when my slide mentions work that you authored

Before We Start

I recently read that . . .

it is not healthy to sit down all day

I propose that you . . .

stand up when my slide mentions work that you authored

let me illustrate that . . .

Before We Start

I recently read that . . .

it is not healthy to sit down all day

I propose that you . . .

stand up when my slide mentions work that you authored

let me illustrate that . . .

Reasoning about Edits to Feature Models

Thomas Thüm
School of Computer Science
University of Magdeburg
tthuem@st.ovgu.de

Don Batory
Dept. of Computer Science
University of Texas at Austin
batory@cs.utexas.edu

Christian Kästner
School of Computer Science
University of Magdeburg
ckaestne@ovgu.de

Abstract

Features express the variabilities and commonalities among programs in a software product line (SPL). A feature model defines the valid combinations of features, where each combination corresponds to a program in an SPL. SPLs and their feature models evolve over time. We classify the evolution of a feature model via modifications as refactorings, specializations, generalizations, or arbitrary edits. We present an algorithm to reason about feature model edits to help designers determine how the program membership of an SPL has changed. Our algorithm takes two feature models as input (before and after edit versions), where the set of features in both models are not necessarily the same, and it automatically computes the change classification. Our algorithm is able to give examples of added or deleted products and efficiently classifies edits to even large models that have thousands of features.

1 Introduction

Software product line (SPL) engineering is a paradigm for systematic reuse [2, 32, 21]. From common assets, different programs of a domain can be assembled. Programs of an SPL are distinguished by *features*, which are domain abstractions relevant to stakeholders and are typically increments in program functionality. Every program of an SPL is represented by a unique combination of features, and an SPL could have millions of distinct programs. A *feature model* compactly defines all features in an SPL and their valid combinations; it is basically an AND-OR graph with constraints.

SPLs and their feature models evolve over time. Even small changes to a feature model, like moving a feature from one branch to another, can unintentionally change the set of legal feature combinations. For example, edits may preclude the creation of programs that were previously built or enlarge the set of programs that can be built. Understanding the impact of feature model edits is known to be impractical to determine manually. Tool support to provide feedback to engineers to explain how changes to a feature model have altered the program membership of an SPL is a fundamental problem.

In prior work, Czarnecki et al. defined operations to specialize feature models [13, 14, 25], where a feature model X is a *specialization* of a feature model Y if the set of products in X is a subset of that in Y. Alves et al. discussed operations called *refactorings* that maintain the set of products or add new products to an SPL [1]. We call this latter case (when new products are added) a *generalization* of a feature model. In both approaches, a set of sound operations is used to edit feature models to determine whether a model is a specialization or refactoring or generalization of another. To determine the effects of feature model edits using these results requires that all edits be expressed as a sequence of predefined operations (refactorings, generalizations, or specializations). However, a tool that allows only sound operations to be performed would have the feel of a hard-to-use structure editor, e.g., how one changes a feature model X into a target feature model Y using a sequence of sound operations is not obvious. A path planner [17] would be needed to automatically compute a sequence (if it exists). If a required operation is missing, designers would be out of luck, as the desired change to a feature model would be precluded by the editor. This approach seems too difficult.

In other work, Sun et al. used first-order logic to determine if two feature models are equivalent, i.e., both models have the same set of products [37]. Janota and Kiniry used higher-order logic to determine whether a feature model is a specialization of another [20]. Both approaches work if both models have the same set of features; their results are not applicable to model edits that add or delete features. Further, we are unaware of performance studies that demonstrate first-order and higher-order logic approaches scale to large feature models efficiently.

In this paper, we present and evaluate an algorithm to determine the relationship between two feature models (i.e., specialization/refactoring/generalization/none of these) using satisfiability solvers. We overcome the limitations of

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

254

Thüm et al. ICSE09

Let Us Do A Dry Run

Reasoning about Edits to Feature Models

Thomas Thüm
School of Computer Science
University of Magdeburg
tthuem@st.ovgu.de

Don Batory
Dept. of Computer Science
University of Texas at Austin
batory@cs.utexas.edu

Christian Kästner
School of Computer Science
University of Magdeburg
ckaestne@ovgu.de

Abstract

Features express the variabilities and commonalities among programs in a software product line (SPL). A feature model defines the valid combinations of features, where each combination corresponds to a program in an SPL. SPLs and their feature models evolve over time. We classify the evolution of a feature model via modifications as refactorings, specializations, generalizations, or arbitrary edits. We present an algorithm to reason about feature model edits to help designers determine how the program membership of an SPL has changed. Our algorithm takes two feature models as input (before and after edit versions), where the set of features in both models are not necessarily the same, and it automatically computes the change classification. Our algorithm is able to give examples of added or deleted products and efficiently classifies edits to even large models that have thousands of features.

1 Introduction

Software product line (SPL) engineering is a paradigm for systematic reuse [2, 32, 21]. From common assets, different programs of a domain can be assembled. Programs of an SPL are distinguished by *features*, which are domain abstractions relevant to stakeholders and are typically increments in program functionality. Every program of an SPL is represented by a unique combination of features, and an SPL could have millions of distinct programs. A *feature model* compactly defines all features in an SPL and their valid combinations; it is basically an AND-OR graph with constraints.

SPLs and their feature models evolve over time. Even small changes to a feature model, like moving a feature from one branch to another, can unintentionally change the set of legal feature combinations. For example, edits may preclude the creation of programs that were previously built or enlarge the set of programs that can be built. Understanding the impact of feature model edits is known to be impractical to determine manually. Tool support to provide feedback to engineers to explain how changes to a feature model have altered the program membership of an SPL is a fundamental problem.

In prior work, Czarnecki et al. defined operations to specialize feature models [13, 14, 25], where a feature model *X* is a *specialization* of a feature model *Y* if the set of products in *X* is a subset of that in *Y*. Alves et al. discussed operations called *refactorings* that maintain the set of products or add new products to an SPL [1]. We call this latter case (when new products are added) a *generalization* of a feature model. In both approaches, a set of sound operations is used to edit feature models to determine whether a model is a specialization or refactoring or generalization of another. To determine the effects of feature model edits using these results requires that all edits be expressed as a sequence of predefined operations (refactorings, generalizations, or specializations). However, a tool that allows only sound operations to be performed would have the feel of a hard-to-use structure editor, e.g., how one changes a feature model *X* into a target feature model *Y* using a sequence of sound operations is not obvious. A path planner [17] would be needed to automatically compute a sequence (if it exists). If a required operation is missing, designers would be out of luck, as the desired change to a feature model would be precluded by the editor. This approach seems too difficult.

In other work, Sun et al. used first-order logic to determine if two feature models are equivalent, i.e., both models have the same set of products [37]. Janota and Kiniry used higher-order logic to determine whether a feature model is a specialization of another [20]. Both approaches work if both models have the same set of features; their results are not applicable to model edits that add or delete features. Further, we are unaware of performance studies that demonstrate first-order and higher-order logic approaches scale to large feature models efficiently.

In this paper, we present and evaluate an algorithm to determine the relationship between two feature models (i.e., specialization/refactoring/generalization/none of these) using satisfiability solvers. We overcome the limitations of

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

Thüm et al. ICSE09

Let Us Do A Dry Run

Reasoning about Edits to Feature Models

Thomas Thüm
School of Computer Science
University of Magdeburg
tthuem@st.ovgu.de

Don Batory
Dept. of Computer Science
University of Texas at Austin
batory@cs.utexas.edu

Christian Kästner
School of Computer Science
University of Magdeburg
ckaestne@ovgu.de

Abstract

Features express the variabilities and commonalities among programs in a software product line (SPL). A feature model defines the valid combinations of features, where each combination corresponds to a program in an SPL. SPLs and their feature models evolve over time. We classify the evolution of a feature model via modifications as refactorings, specializations, generalizations, or arbitrary edits. We present an algorithm to reason about feature model edits to help designers determine how the program membership of an SPL has changed. Our algorithm takes two feature models as input (before and after edit versions), where the set of features in both models are not necessarily the same, and it automatically computes the change classification. Our algorithm is able to give examples of added or deleted products and efficiently classifies edits to even large models that have thousands of features.

1 Introduction

Software product line (SPL) engineering is a paradigm for systematic reuse [2, 32, 21]. From common assets, different programs of a domain can be assembled. Programs of an SPL are distinguished by *features*, which are domain abstractions relevant to stakeholders and are typically increments in program functionality. Every program of an SPL is represented by a unique combination of features, and an SPL could have millions of distinct programs. A *feature model* compactly defines all features in an SPL and their valid combinations; it is basically an AND-OR graph with constraints.

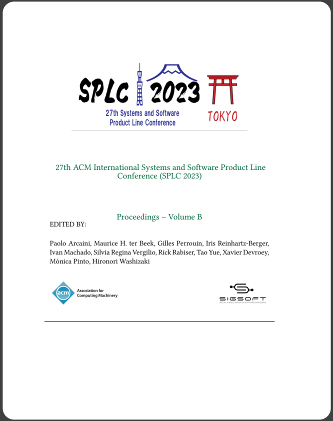
SPLs and their feature models evolve over time. Even small changes to a feature model, like moving a feature from one branch to another, can unintentionally change the set of legal feature combinations. For example, edits may preclude the creation of programs that were previously built or enlarge the set of programs that can be built. Understanding the impact of feature model edits is known to be impractical to determine manually. Tool support to provide feedback to engineers to explain how changes to a feature model have altered the program membership of an SPL is a fundamental problem.

In prior work, Czarnecki et al. defined operations to specialize feature models [13, 14, 25], where a feature model X is a *specialization* of a feature model Y if the set of products in X is a subset of that in Y. Alves et al. discussed operations called *refactorings* that maintain the set of products or add new products to an SPL [1]. We call this latter case (when new products are added) a *generalization* of a feature model. In both approaches, a set of sound operations is used to edit feature models to determine whether a model is a specialization or refactoring or generalization of another. To determine the effects of feature model edits using these results requires that all edits be expressed as a sequence of predefined operations (refactorings, generalizations, or specializations). However, a tool that allows only sound operations to be performed would have the feel of a hard-to-use structure editor, e.g., how one changes a feature model X into a target feature model Y using a sequence of sound operations is not obvious. A path planner [17] would be needed to automatically compute a sequence (if it exists). If a required operation is missing, designers would be out of luck, as the desired change to a feature model would be precluded by the editor. This approach seems too difficult.

In other work, Sun et al. used first-order logic to determine if two feature models are equivalent, i.e., both models have the same set of products [37]. Janota and Kiniry used higher-order logic to determine whether a feature model is a specialization of another [20]. Both approaches work if both models have the same set of features; their results are not applicable to model edits that add or delete features. Further, we are unaware of performance studies that demonstrate first-order and higher-order logic approaches scale to large feature models efficiently.

In this paper, we present and evaluate an algorithm to determine the relationship between two feature models (i.e., specialization/refactoring/generalization/none of these) using satisfiability solvers. We overcome the limitations of

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE



27th ACM International Systems and Software Product Line Conference (SPLC 2023)

Proceedings – Volume B

EDITED BY:

Paolo Arcaini, Maurice H. ter Beek, Gilles Perrouin, Iris Reinhartz-Berger, Ivan Machado, Silvia Regina Vergilio, Rick Rabiser, Tao Yue, Xavier Devroey, Mónica Pinto, Hironori Washizaki

Thüm et al. ICSE09

SPLC 2023
Proceedings — Volume B

Let Us Do A Dry Run

Reasoning about Edits to Feature Models

Thomas Thüm, School of Computer Science, University of Magdeburg, tthuem@st.ovgu.de
Don Batory, Dept. of Computer Science, University of Texas at Austin, batory@cs.utexas.edu
Christian Kästner, School of Computer Science, University of Magdeburg, ckaestne@ovgu.de

Abstract

Features express the variabilities and commonalities among programs in a software product line (SPL). A feature model defines the valid combinations of features, where each combination corresponds to a program in an SPL. SPLs and their feature models evolve over time. We classify the evolution of a feature model via modifications as refactorings, specializations, generalizations, or arbitrary edits. We present an algorithm to reason about feature model edits to help designers determine how the program membership of an SPL has changed. Our algorithm takes two feature models as input (before and after edit versions), where the set of features in both models are not necessarily the same, and it automatically computes the change classification. Our algorithm is able to give examples of added or deleted products and efficiently classifies edits to even large models that have thousands of features.

1 Introduction

Software product line (SPL) engineering is a paradigm for systematic reuse [2, 32, 21]. From common assets, different programs of a domain can be assembled. Programs of an SPL are distinguished by *features*, which are domain abstractions relevant to stakeholders and are typically increments in program functionality. Every program of an SPL is represented by a unique combination of features, and an SPL could have millions of distinct programs. A *feature model* compactly defines all features in an SPL and their valid combinations; it is basically an AND-OR graph with constraints.

SPLs and their feature models evolve over time. Even small changes to a feature model, like moving a feature from one branch to another, can unintentionally change the set of legal feature combinations. For example, edits may preclude the creation of programs that were previously built or enlarge the set of programs that can be built. Understanding the impact of feature model edits is known to be impractical to determine manually. Tool support to provide feedback to engineers to explain how changes to a feature model have altered the program membership of an SPL is a fundamental problem.

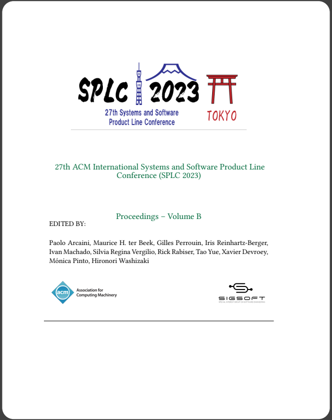
In prior work, Czarnecki et al. defined operations to specialize feature models [13, 14, 25], where a feature model X is a *specialization* of a feature model Y if the set of products in X is a subset of that in Y. Alves et al. discussed operations called *refactorings* that maintain the set of products or add new products to an SPL [1]. We call this latter case (when new products are added) a *generalization* of a feature model. In both approaches, a set of sound operations is used to edit feature models to determine whether a model is a specialization or refactoring or generalization of another. To determine the effects of feature model edits using these results requires that all edits be expressed as a sequence of predefined operations (refactorings, generalizations, or specializations). However, a tool that allows only sound operations to be performed would have the feel of a hard-to-use structure editor, e.g., how one changes a feature model X into a target feature model Y using a sequence of sound operations is not obvious. A path planner [17] would be needed to automatically compute a sequence (if it exists). If a required operation is missing, designers would be out of luck, as the desired change to a feature model would be precluded by the editor. This approach seems too difficult.

In other work, Sun et al. used first-order logic to determine if two feature models are equivalent, i.e., both models have the same set of products [37]. Janota and Kiniry used higher-order logic to determine whether a feature model is a specialization of another [20]. Both approaches work if both models have the same set of features; their results are not applicable to model edits that add or delete features. Further, we are unaware of performance studies that demonstrate first-order and higher-order logic approaches scale to large feature models efficiently.

In this paper, we present and evaluate an algorithm to determine the relationship between two feature models (i.e., specialization/refactoring/generalization/none of these) using satisfiability solvers. We overcome the limitations of

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE 254

Thüm et al. ICSE09



SPLC 2023
27th Systems and Software Product Line Conference
TOKYO

27th ACM International Systems and Software Product Line Conference (SPLC 2023)

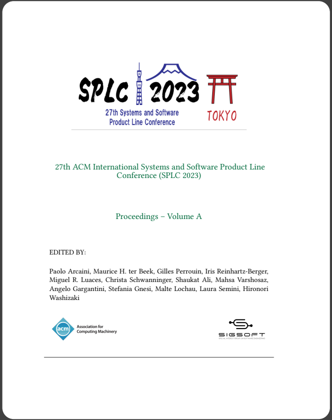
Proceedings – Volume B

EDITED BY:

Paolo Arcaini, Maurice H. ter Beek, Gilles Perrouin, Iris Reinhartz-Berger, Ivan Machado, Silvia Regina Vergilio, Rick Rabiser, Tao Yue, Xavier Devroey, Mónica Pinto, Hironori Washizaki

Association for Computing Machinery
SIGSOFT

SPLC 2023
Proceedings — Volume B



SPLC 2023
27th Systems and Software Product Line Conference
TOKYO

27th ACM International Systems and Software Product Line Conference (SPLC 2023)

Proceedings – Volume A

EDITED BY:

Paolo Arcaini, Maurice H. ter Beek, Gilles Perrouin, Iris Reinhartz-Berger, Miguel R. Luaces, Christa Schwanninger, Shaukat Ali, Mahsa Varshosaz, Angelo Gargantini, Stefania Gnesi, Malte Lochau, Laura Semini, Hironori Washizaki

Association for Computing Machinery
SIGSOFT

SPLC 2023
Proceedings — Volume A

What Motivated This Work?

Let's Travel Back to 2007

First iPhone Released

3.5 in display with 480x320 pixel resolution

Let's Travel Back to 2007

First iPhone Released

3.5 in display with 480x320 pixel resolution

Thomas in a Mexican Shop in Texas

full resolution without glasses

Let's Travel Back to 2007

First iPhone Released

no front cam, no bluetooth, no wireless charging, no AI

Thomas in Texas

no bachelor, no master, no PhD, no tenure

My Bachelor's Thesis

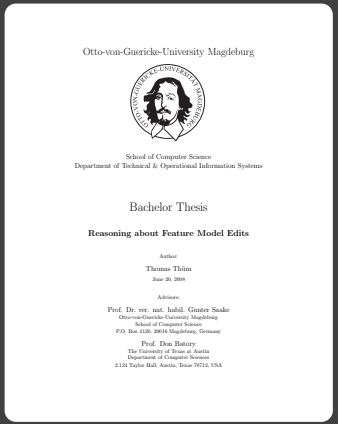
"Hey Christian, I want to do my bachelor's thesis abroad!"

My Bachelor's Thesis

"Hey Christian, I want to do my bachelor's thesis abroad!"

"Hey Don, what can I work on for my bachelor's thesis?"

My Bachelor's Thesis



Otto-von-Guericke-University Magdeburg

School of Computer Science
Department of Technical & Operational Information Systems

Bachelor Thesis

Reasoning about Feature Model Edits

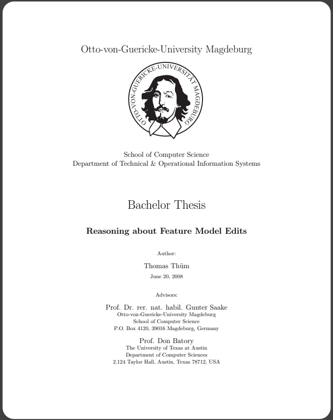
Author:
Thomas Thüm
June 20, 2008

Advisors:
Prof. Dr. rer. nat. habil. Gunter Saake
Otto-von-Guericke-University Magdeburg
School of Computer Science
P.O. Box 4120, 39016 Magdeburg, Germany

Prof. Don Batory
The University of Texas at Austin
Department of Computer Sciences
2.124 Taylor Hall, Austin, Texas 78712, USA

submitted June 2008

My Bachelor's Thesis



Otto-von-Guericke-University Magdeburg

School of Computer Science
Department of Technical & Operational Information Systems

Bachelor Thesis

Reasoning about Feature Model Edits

Author:

Thomas Thüm

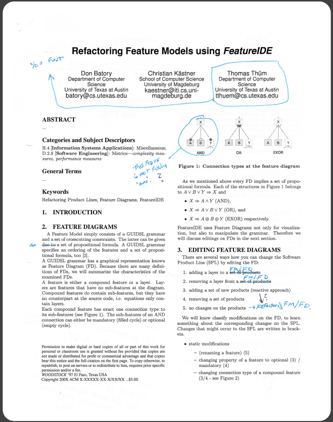
June 20, 2008

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Otto-von-Guericke-University Magdeburg
School of Computer Science
P.O. Box 4120, 39016 Magdeburg, Germany

Prof. Don Batory
The University of Texas at Austin
Department of Computer Sciences
2.124 Taylor Hall, Austin, Texas 78712, USA

submitted June 2008



Refactoring Feature Models using *FeatureIDE*

Don Batory, Department of Computer Science, University of Texas at Austin, batory@cs.utexas.edu
Christian Kästner, School of Computer Science, University of Magdeburg, kaestner@iti.cs.uni-magdeburg.de
Thomas Thüm, Department of Computer Science, University of Texas at Austin, tthuem@cs.utexas.edu

ABSTRACT

...

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

...

Keywords

Refactoring Product Lines, Feature Diagrams, FeatureIDE

1. INTRODUCTION

2. FEATURE DIAGRAMS

A Feature Model simply consists of a GUIDSL grammar and a set of crosscutting constraints. The latter can be given ~~due to~~ a set of propositional formula. A GUIDSL grammar specifies an ordering of the features and a set of propositional formula, too [2].
A GUIDSL grammar has a graphical representation known as Feature Diagram (FD). Because there are many definitions of FDs, we will summarize the characteristics of the examined FDs.
A feature is either a compound feature or a layer. Layers are features that have no sub-features at the diagram. Compound features do contain sub-features, but they have no counterpart at the source code, i.e. equations only contain layers.
Each compound feature has exact one connection type to its sub-features (see Figure 1). The sub-features of an AND connection can either be mandatory (filled cycle) or optional (empty cycle).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WOODSTOCK '97 El Paso, Texas USA
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

X, A, B, Y — AND, OR, EXOR

Figure 1: Connection types at the feature diagram

As we mentioned above every FD implies a set of propositional formula. Each of the structures in Figure 1 belongs to $A \vee B \vee Y \Rightarrow X$ and

- $X \Rightarrow A \wedge Y$ (AND),
- $X \Rightarrow A \vee B \vee Y$ (OR), and
- $X \Rightarrow A \oplus B \oplus Y$ (EXOR) respectively.

FeatureIDE uses Feature Diagrams not only for visualization, but also to manipulate the grammar. Therefore we will discuss editings on FDs in the next section.

3. EDITING FEATURE DIAGRAMS

There are several ways how you can change the Software Product Line (SPL) by editing the FD:

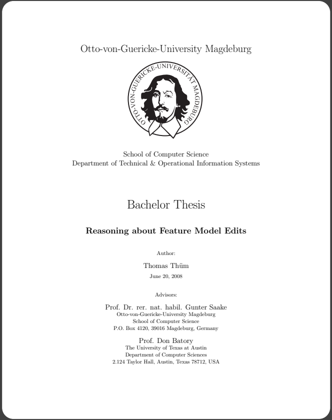
1. adding a layer to a ~~set of products~~
2. removing a layer from a ~~set of products~~
3. adding a set of new products (reactive approach)
4. removing a set of products
5. no changes on the products

We will know classify modifications on the FD, to learn something about the corresponding changes on the SPL. Changes that might occur to the SPL are written in brackets.

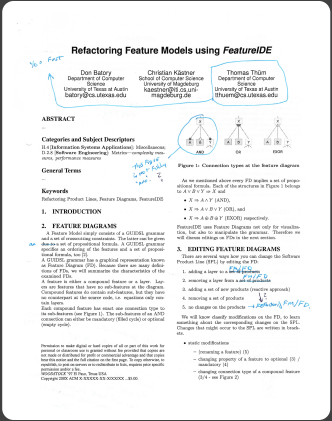
- static modifications
  - (renaming a feature) (5)
  - changing property of a feature to optional (3) / mandatory (4)
  - changing connection type of a compound feature (3/4 - see Figure 2)

started to write a paper

My Bachelor's Thesis



submitted June 2008



started to write a paper

submitted August 2008

Reasoning about Edits to Feature Models

Thomas Thüm
University of Magdeburg
Germany

Don Batory
University of Texas
USA

Christian Kästner
University of Magdeburg
Germany

all historic slides
linked on Twitter/X

Reasoning about Edits to Feature Models

Thomas Thüm
University of Magdeburg
Germany

Don Batory
University of Texas
USA

Christian Kästner
University of Magdeburg
Germany

all historic slides linked on Twitter/X

we were lucky! (12 % accept. rate)

only five earlier ICSE papers on product lines

I was lucky! (to work with Don and Christian)

Customizable Software

Movements in software development

1. All-in-one software suitable for many purposes
2. Customized software for special requirements

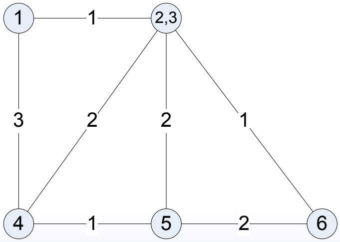
Software product line (SPL)

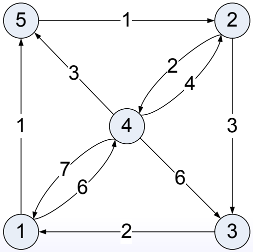
- Set of software-intensive systems that share a common, managed set of features
- Reuse of development artifacts
- Variability commonly expressed by feature models

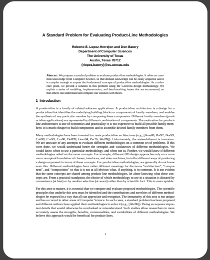
clear to you :)

Customizable Graph Library

- Undirected and directed graphs
- Does the graph contain a cycle?
- What is the number of edges?
- Find a shortest path between two given nodes
- Customers have different needs






A Standard Problem for Evaluating Product-Line Methodologies

Roberto E. Lopez-Herrejon and Don Batory
Department of Computer Sciences
The University of Texas
Austin, Texas 78712
{rlopez,batory}@cs.utexas.edu

Abstract. We propose a standard problem to evaluate product-line methodologies. It relies on common knowledge from Computer Science, so that domain-knowledge can be easily acquired, and it is complex enough to expose the fundamental concepts of product-line methodologies. As a reference point, we present a solution to this problem using the GenVoca design methodology. We explain a series of modeling, implementation, and benchmarking issues that we encountered, so that others can understand and compare our solution with theirs.

1 Introduction

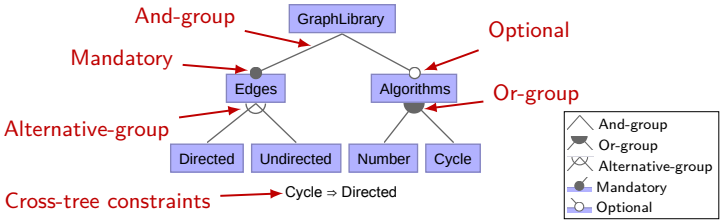
A *product-line* is a family of related software applications. A *product-line architecture* is a design for a product-line that identifies the underlying building blocks or components of family members, and enables the synthesis of any particular member by composing these components. Different family members (product-line applications) are represented by different combination of components. The motivation for product-line architectures is one of economics and practicality: it is too expensive to build all possible family members; it is much cheaper to build components and to assemble desired family members from them.

Lopez-Herrejon and Batory 2001

(graph product line)

Feature Models

- Features used to distinguish program variants (Kang et al. 90)
- Feature models specify SPL features and their combinations
- Graphically represented by feature diagrams:



- Valid feature selections (products):

  $\{G, E, D\}$, $\{G, E, U\}$, $\{G, E, D, A, N\}$, $\{G, E, D, A, C\}$, $\{G, E, U, A, N\}$, $\{G, E, D, A, N, C\}$

- In practice: hundreds of features, millions of products

also clear to you :)

but this is our running example

Feature Models Evolve

Software product lines and their feature models evolve over time

Basic edits to feature models:

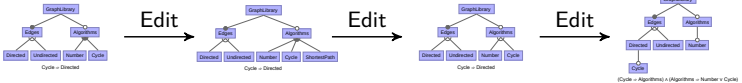
1. Changing a group type
2. Changing optional feature to mandatory or vice versa
3. Adding/removing a feature
4. Adding/removing a constraint
5. Moving a feature

Edit categories:

- Support domain engineers when editing feature models
- Guarantee that no products are added or deleted or both

Feature Models Evolve

Software product lines and their feature models evolve over time



Basic edits to feature models:

1. Changing a group type
2. Changing optional feature to mandatory or vice versa
3. Adding/removing a feature
4. Adding/removing a constraint
5. Moving a feature

Edit categories:

- Support domain engineers when editing feature models
- Guarantee that no products are added or deleted or both

Kröher et al. JSS23

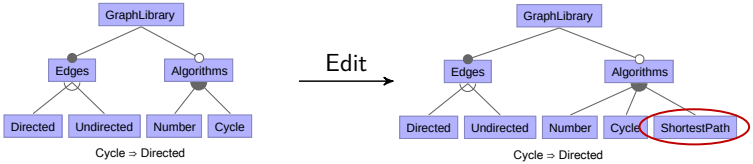
4 % of commits in Linux kernel affect feature model

1 % solely change the feature model

What Is Our Contribution?

Generalization

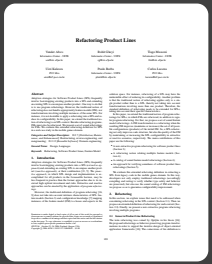
- Adds new products to an SPL
- Examples: new features, less constraints



$\{G, E, D\}$, $\{G, E, U\}$,
$\{G, E, D, A, N\}$, $\{G, E, D, A, C\}$,
$\{G, E, U, A, N\}$, $\{G, E, D, A, N, C\}$

$\subset$

$\{G, E, D\}$, $\{G, E, U\}$,
$\{G, E, D, A, N\}$, $\{G, E, D, A, C\}$,
$\{G, E, D, A, S\}$, $\{G, E, U, A, N\}$,
$\{G, E, U, A, S\}$, $\{G, E, U, A, N, C\}$,
$\{G, E, D, A, N, S\}$,
$\{G, E, D, A, C, S\}$,
$\{G, E, U, A, N, S\}$,
$\{G, E, D, A, N, C, S\}$


Refactoring Product Lines

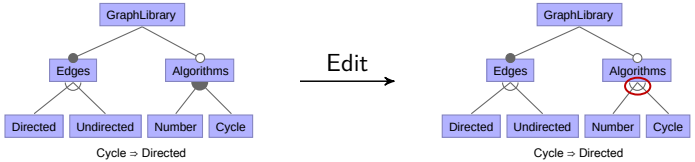
Alves et al. GPCE06

(edit patterns)

motivation for our work

Specialization

- Removes products from an SPL
- Examples: removed features, additional constraints, staged configuration



$\{G, E, D\}$, $\{G, E, U\}$,
$\{G, E, D, A, N\}$, $\{G, E, D, A, C\}$,
$\{G, E, U, A, N\}$, $\{G, E, D, A, N, C\}$

$\supset$

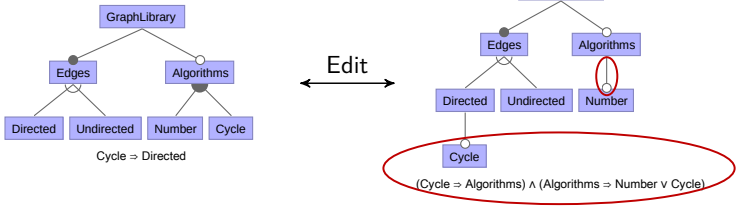
$\{G, E, D\}$, $\{G, E, U\}$,
$\{G, E, D, A, N\}$, $\{G, E, D, A, C\}$,
$\{G, E, U, A, N\}$

Czarnecki et al.
SPIP05

(staged configuration)

Refactoring

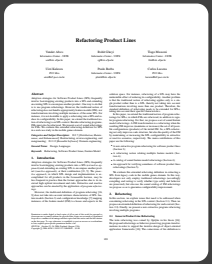
- Changes the feature model without affecting the SPL
- Useful to improve readability, maintainability, and extensibility
- Examples: moving features, rewriting constraints



$\{G, E, D\}, \{G, E, U\},$
$\{G, E, D, A, N\}, \{G, E, D, A, C\},$
$\{G, E, U, A, N\}, \{G, E, D, A, N, C\}$

$=$

$\{G, E, D\}, \{G, E, U\},$
$\{G, E, D, A, N\}, \{G, E, D, A, C\},$
$\{G, E, U, A, N\}, \{G, E, D, A, N, C\}$

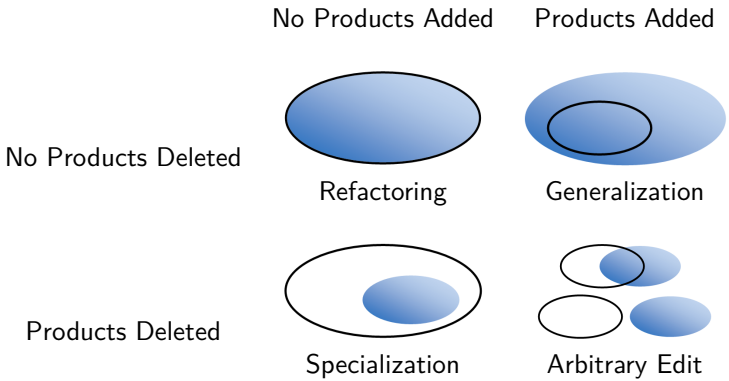

Refactoring Product Lines

Alves et al. GPCE06

(edit patterns)

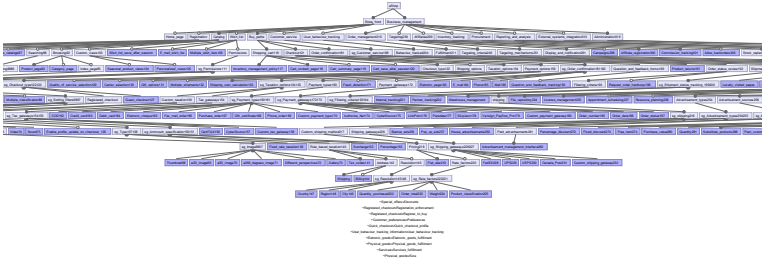
motivation for our work

Categories of Edits



Our Goal

1. Automatically categorize an edit even for large feature models
2. Calculate examples for added and deleted products if available



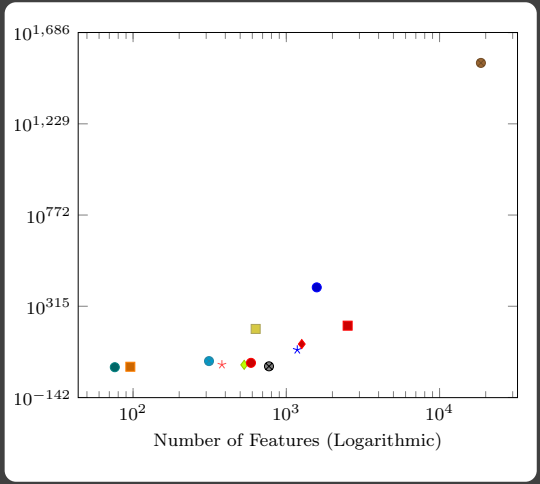
Mendonca, University of Waterloo

Easy to determine for small feature models, but a matter of scale!

How Large Are Configuration Spaces?

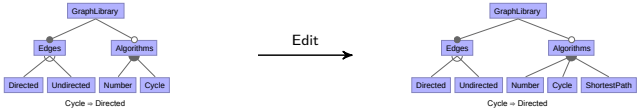
Sundermann et al.
EMSE23

(size of configuration spaces)



1. Enumeration

Determine and compare the set of all products for both software product lines



$\{G, E, D\}$, $\{G, E, U\}$,
$\{G, E, D, A, N\}$, $\{G, E, D, A, C\}$,
$\{G, E, U, A, N\}$, $\{G, E, D, A, N, C\}$

?

$\{G, E, D\}$, $\{G, E, U\}$,
$\{G, E, D, A, N\}$, $\{G, E, D, A, C\}$,
$\{G, E, D, A, S\}$, $\{G, E, U, A, N\}$,
$\{G, E, U, A, S\}$, $\{G, E, U, A, N, C\}$,
$\{G, E, D, A, N, S\}$,
$\{G, E, D, A, C, S\}$,
$\{G, E, U, A, N, S\}$,
$\{G, E, D, A, N, C, S\}$

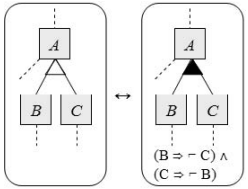
Enumeration does not scale!

Sundermann et al. EMSE23

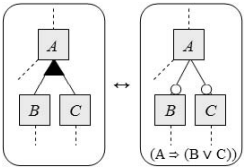
(size of configuration spaces)

2. Sound Operations

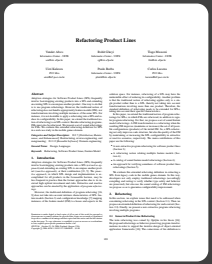
- Introduced by Alves et al. in 2006
- Catalogue of operations that are known to be a refactoring
- Catalogues for generalization and specialization



(a) *Replace Alternative* Refactoring



(b) *Replace Or* Refactoring



Alves et al. GPCE06

(edit patterns)

motivation for our work

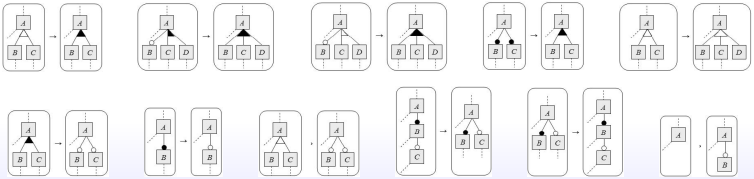
2. Sound Operations

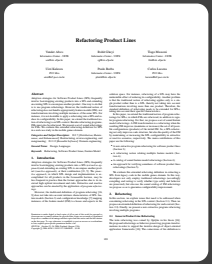
Advantages:

- Intuitive: only special operations are allowed

Disadvantages:

- How to compare feature models build from scratch?
- What if we want to use edits not in the operation set, e.g., moving a feature?
- Requires hard-to-use structural editors or path-finder algorithms
- Different catalogues necessary for other variability models





Refactoring Product Lines

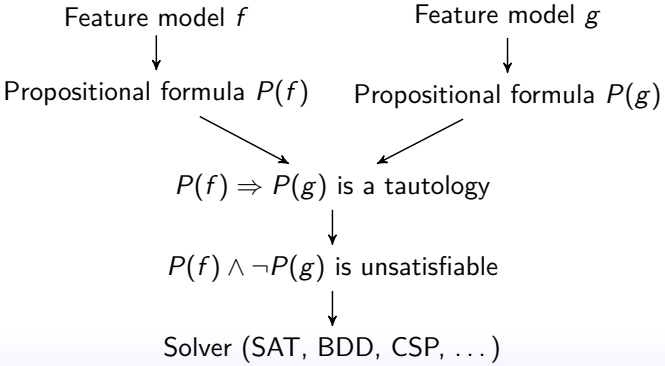
Alves et al. GPCE06

(edit patterns)

motivation for our work

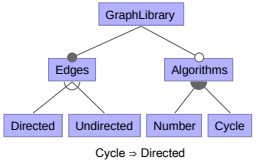

3. Reasoning using Propositional Formulas

- Proposed by Sun et al. in 2005, Janota and Kiniri in 2007
- Are all products from feature model f available in g?



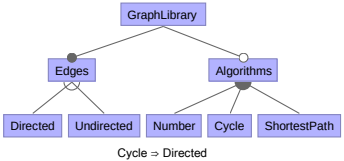
3. Reasoning using Propositional Formulas

Standard translation into a propositional formula (Batory 05)



$$
\begin{aligned}
&(G \wedge \\
&(E \Rightarrow G) \wedge (A \Rightarrow G) \wedge (G \Rightarrow E) \wedge \\
&(E \Rightarrow D \vee U) \wedge (\neg D \vee \neg U) \wedge \\
&(A \Rightarrow N \vee C) \wedge (N \Rightarrow A) \wedge \\
&(C \Rightarrow A) \wedge \\
&(C \Rightarrow D))
\end{aligned}
$$

$\bigwedge\neg$



$$
\begin{aligned}
&(G \wedge \\
&(E \Rightarrow G) \wedge (A \Rightarrow G) \wedge (G \Rightarrow E) \wedge \\
&(E \Rightarrow D \vee U) \wedge (\neg D \vee \neg U) \wedge \\
&(A \Rightarrow N \vee C \vee S) \wedge (N \Rightarrow A) \wedge \\
&(C \Rightarrow A) \wedge (S \Rightarrow A) \wedge \\
&(C \Rightarrow D))
\end{aligned}
$$

The propositional formula above is satisfiable if and only if the edit deletes products

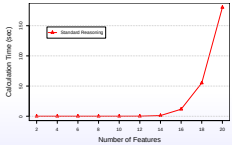
3. Reasoning using Propositional Formulas

Advantages:

- Cross-tree constraints can be altered arbitrary
- Edits might be unknown
- All edits can be classified
- Applicable to all variability models that can be represented as a propositional formula
- Variability models of different types can be compared

Disadvantages:

- Restricted to feature models with the same feature set
- Performance!



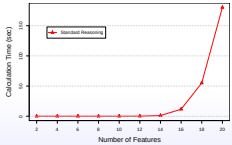
3. Reasoning using Propositional Formulas

Advantages:

- Cross-tree constraints can be altered arbitrary
- Edits might be unknown
- All edits can be classified
- Applicable to all variability models that can be represented as a propositional formula
- Variability models of different types can be compared

Disadvantages:

- Restricted to feature models with the same feature set
- Performance!



very bad performance

+

I received very critical feedback to the concept chapter of my bachelor's thesis . . .

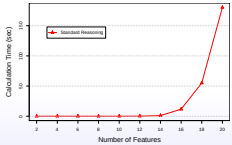
3. Reasoning using Propositional Formulas

Advantages:

- Cross-tree constraints can be altered arbitrary
- Edits might be unknown
- All edits can be classified
- Applicable to all variability models that can be represented as a propositional formula
- Variability models of different types can be compared

Disadvantages:

- Restricted to feature models with the same feature set
- Performance!



very bad performance

+

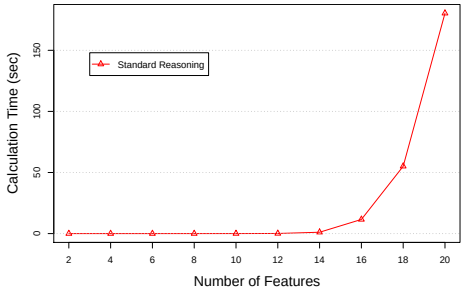
I received very critical feedback to the concept chapter of my bachelor's thesis . . .

"shall I quit my bachelor?"

"I could work as a programmer in industry . . . "


BUT NOOOOO!

3. Reasoning using Propositional Formulas



- Off-the-shelf solvers need propositional formulas to be in conjunctive normal form (CNF)
- $P(f) \wedge \neg P(g)$ needs to be converted into CNF
- Converting $P(f)$ is easy because the structure of feature models is CNF-like
- But: the negation of $P(g)$ is very time consuming

Bittner et al. SEFM19

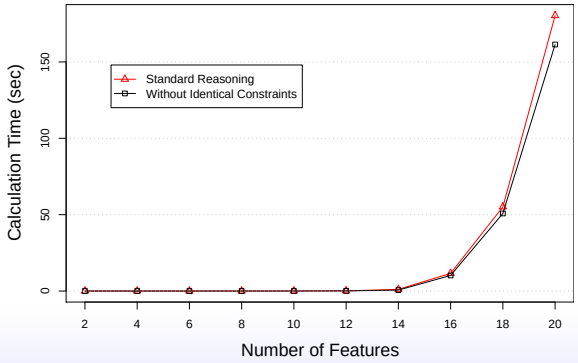
Kuiter et al. ASE22

translation to CNF is not always easy

Simplified Reasoning - First Optimization

Removing identical constraints:

- $P(f) \wedge \neg P(g) \equiv P(f) \wedge \neg p_g$, where $P(g) = p_g \wedge c$ and c are unchanged rules
- The formula to solve is smaller

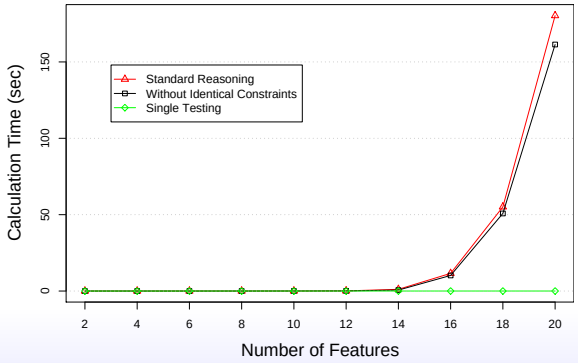


details in the paper

Simplified Reasoning - Second Optimization

Single testing:

- $P(f) \wedge \neg p_g \equiv P(f) \wedge \neg \bigwedge_{1 \leq i \leq n'} R_i \equiv \bigvee_{1 \leq i \leq n'} (P(f) \wedge \neg R_i)$
- Splitting formula in multiple small SAT problems reduces calculation time

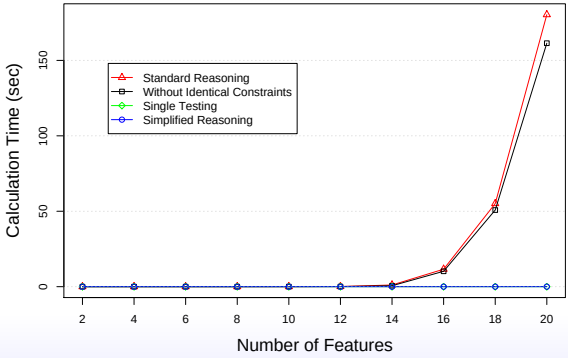


the core contribution!

Simplified Reasoning - Third Optimization

Stop early:

- Stop if a satisfiable formula $P(f) \wedge \neg R_i$ is found
- Saves calculation time when many edits were applied



details in the paper

Simplified Reasoning - Extensions

Special handling for added/removed features

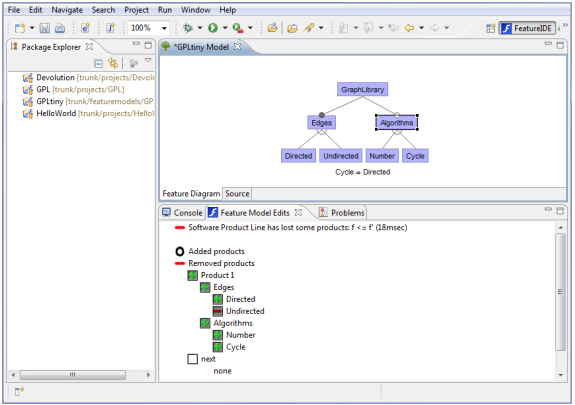
- Added features are not selectable in the old feature model version
- Removed features are not selectable in the new feature model version

Special handling for abstract features

- Abstract features are features that do not change the implementation
- Example: removing an abstract feature is a refactoring
- See the paper for details

details in the paper

Implemented in FeatureIDE



Formal tool demonstration tomorrow at 4:30pm

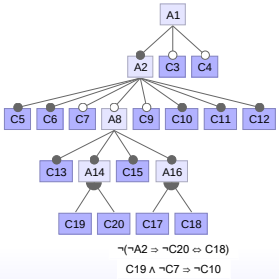
Available open source at `http://www.fosd.de/featureide`

sorry for increasing YOUR environmental footprint

Does It Scale?

Does Our Approach Scale?

- Do our optimizations allow reasoning about large feature models within reasonable time?
- Unable to aquire sufficient amount of large feature models
- We generated 2000 feature models resembling feature models from practice

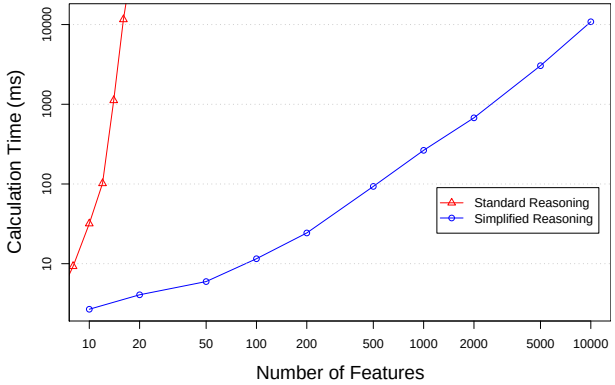


Feature diagram probabilities:
* And-group: 50%
* Or-group: 25%
* Alternative-group: 25%
* Optional child: 50%
* Maximum number of children: 10

Cross-tree constraints:
* 10% from number of features
* Between 2 and 5 variables

All 2000 generated feature models are avaible on FeatureIDE's website for comparative studies.

Scalability - Number of Features



- We applied 10 edits to each feature model
- Calculation time to classify edits increases almost linearly

optimizations are crucial

What Is The Impact?

Disclaimer: expect mistakes in the next slides

Remarks on Soundness

Acher et al. SLE09

"The classification proposed in [MIP] covers all the changes a designer can produce on a FM and the formalization provided in [MIP] is a sound basis for reasoning about these changes."

Remarks on Soundness

Acher et al. SLE09

"The classification proposed in [MIP] covers all the changes a designer can produce on a FM and the formalization provided in [MIP] is a sound basis for reasoning about these changes."

Berger et al. TSE13

"Assumptions in the literature about content, structure, and constraints of models differ from our results. [...] Thüm et al. generate trees with maximal branching factors of 10 (too low, see Section 5.2.2), with 25 percent of inner features representing OR groups (too high, see Section 5.3.1), ~~and 10 percent of all features having additional constraints (too low, see Section 5.3.2)~~. [...] Our results challenge all these assumptions."

Remarks on Soundness

Acher et al. SLE09

"The classification proposed in [MIP] covers all the changes a designer can produce on a FM and the formalization provided in [MIP] is a sound basis for reasoning about these changes."

Berger et al. TSE13

"Assumptions in the literature about content, structure, and constraints of models differ from our results. [...] Thüm et al. generate trees with maximal branching factors of 10 (too low, see Section 5.2.2), with 25 percent of inner features representing OR groups (too high, see Section 5.3.1), ~~and 10 percent of all features having additional constraints (too low, see Section 5.3.2)~~. [...] Our results challenge all these assumptions."

Thüm et al. ICSE09

"Our prior concern is whether generated feature models and edits are representative of industrial usage."

Remarks on Novelty and Soundness

Passos et al. EMSE16

"In Thüm's approach, reasoning is performed by efficiently translating both the original variability model and the one resulting from the changes into a satisfiability problem; by avoiding an exponential explosion of CNF clauses, the proposed reasoning has been tested over large models, showing to scale with randomly-generated models with up to 10,000 features. Moreover, reasoning does not require variability models to have the same set of features, as generalization can include new ones, and specialization remove others."

Remarks on Novelty and Soundness

Passos et al. EMSE16

"In Thüm's approach, reasoning is performed by efficiently translating both the original variability model and the one resulting from the changes into a satisfiability problem; by avoiding an exponential explosion of CNF clauses, the proposed reasoning has been tested over large models, showing to scale with randomly-generated models with up to 10,000 features. Moreover, reasoning does not require variability models to have the same set of features, as generalization can include new ones, and specialization remove others."

Passos et al. EMSE16

"Interestingly, existing research has focused much of its efforts on variability evolution as it occurs in the variability model, but it has ignored the coevolution of other related artifacts. Despite the advances of the work of Thüm et al.,

"

Remarks on Novelty and Soundness

Passos et al. EMSE16

"In Thüm's approach, reasoning is performed by efficiently translating both the original variability model and the one resulting from the changes into a satisfiability problem; by avoiding an exponential explosion of CNF clauses, the proposed reasoning has been tested over large models, showing to scale with randomly-generated models with up to 10,000 features. Moreover, reasoning does not require variability models to have the same set of features, as generalization can include new ones, and specialization remove others."

Passos et al. EMSE16

"Interestingly, existing research has focused much of its efforts on variability evolution as it occurs in the variability model, but it has ignored the coevolution of other related artifacts. Despite the advances of the work of Thüm et al., their approach may not produce sound results in the case of changes that affect the feature set, but that preserve the overall functionality of the target software through changes in other spaces. The merging of FB_IMAC into FB_EFI, discussed in Section 2.2, illustrates this situation. However, since the edit-reasoning technique of Thüm et al. considers only changes of the variability model, it would report the discussed merge as specialization, which would be incorrect [...]."

Even More Remarks on Soundness

Abstract Features in Feature Modeling

Thomas Thüm*, Christian Kästner†, Sebastian Erdweg†, and Norbert Siegmund*

*University of Magdeburg, Germany
†Philipps University Marburg, Germany

Abstract—A software product line is a set of program variants, typically generated from a common code base. Feature models describe variability in product lines by enumerating features and their valid combinations. In product-line engineering, we often need to reason about sets of program variants for many different tasks. For example, given a feature model, we might want to determine the number of all valid feature combinations or to select a specific program variant for testing. However, we found that contemporary reasoning approaches can only reason about feature combinations, not about program variants, because they do not properly take feature sets into account. Abstract features are features used to structure a feature model that, however, do not have any impact at implementation level. Using existing feature-model reasoning mechanisms for program variants leads to incorrect results. We propose the notion of abstract features to explicitly express that do not affect the generation of a program variant. We raise awareness of the problem of abstract features for different tasks, such as configuration and testing. To enable correct ways to reason about program variants, abstract features should be made explicit in feature models. We present a technique based on propositional formulas that reasons directly about program variants rather than feature combinations. In practice, our technique can save effort that is caused by considering the same program variant multiple times, for example, in product-line testing.

***Keywords*-Software product lines, program families, feature modeling, feature model, automated analyses.**

I. Introduction

A *software product line* is a set of software-intensive systems (program variants) that share common code artifacts [1]. The overall idea is to efficiently develop similar programs simultaneously by systematically reusing developer artifacts. Program variants are distinguished in terms of *features*, which are units pertinent to stakeholders' concerns, aspects, qualities, or characteristics of a software system [2]. Two variants may have several features in common and differ in other features. In particular, we focus on product-line engineering methods, in which program variants can be *generated* from a common implementation by specifying a selection of features. This includes implementation techniques, such as conditional compilation [3], plug-ins and frameworks, or advanced programming-language mechanisms [4], aspects [4], feature modules [5], [6], or delta modules [7].

In general, not all combinations of features are useful and result in meaningful program variants. For instance, there might be mutually exclusive features or features that require other features. A variability model specifies all valid *feature combinations* and thus the program variants that can be generated. A common form of variability models are feature models [2], [8], [9]. Feature models can have several representations, e.g., a propositional formula in which each satisfying assignment to the formula evaluates to true for all valid combinations [10].

We noticed that not all features in typical feature models are used to distinguish program variants. Some are only used to structure the model and selecting or eliminating them does not make any difference in the generated variant code. We denote such features as *abstract features*. Due to abstract features, the set of *feature combinations* and the set of *program variants* are not equivalent. Multiple valid feature combinations result in the same generated program variant.

There are numerous approaches to reason about valid feature combinations in feature models [9]. However, there are also many interesting questions, for which we want to reason about program variants. For example, for combinatoric testing [11], type checking [12]–[15], or verification [16], we want to select certain sets of program variants or need to reason about the relationship of features in program variants. Finally, when refactoring feature models with respect to the program variants [17], abstract features are not relevant, but can increase measurement effort significantly. Finally, deleting an abstract feature from the feature model does not change possible program variants and may hence be considered as feature-model refactoring [18]. Contemporary automated analyses of feature models can be used to reason about feature combinations, but not directly to reason about program variants. When using automated reasoning nonetheless, the results are at least inaccurate or inefficient.

As a consequence, we distinguish two semantics: (a) the semantics of feature models as known from literature [19], describing the valid combinations of features and (b) the semantics of program families describing distinct program variants of a product line. According to our experience, both are needed and complementary to each other. However, in order to reason about program variants automatically on the program-families semantics, we provide a mechanism to translate the program-families semantics in a way that existing reasoning mechanisms for the feature-model seman-

Thüm et al. SPLC11

Even More Remarks on Soundness

Abstract Features in Feature Modeling

Thomas Thüm*, Christian Kästner†, Sebastian Erdweg†, and Norbert Siegmund*

*University of Magdeburg, Germany
†Philipps University Marburg, Germany

***Abstract*—A software product line is a set of program variants, typically generated from a common code base. Feature models describe variability in product lines by documenting features and their valid combinations. In product-line engineering, we need to reason about variability and program variants for many different tasks. For example, given a feature model, we might want to determine the number of all valid feature combinations or compute specific feature combinations for testing. However, we found that contemporary reasoning approaches can only reason about feature combinations, not about program variants, because they do not take abstract features into account. Abstract features are features used to structure a feature model that, however, do not have any impact at implementation level. Using existing feature-model reasoning mechanisms for program variants leads to incorrect results. Hence, although abstract features represent domain decisions that do not affect the generation of a program variant. We raise awareness of the problem of abstract features for different kinds of analyses on feature models. We argue that, in order to reason about program variants, abstract features should be made explicit in feature models. We present a technique based on propositional formulas that enables to reason about program variants rather than feature combinations. In practice, our technique can save effort that is caused by considering the same program variant multiple times, for example, in product-line testing.**

***Keywords*-Software product lines, program families, feature modeling, feature model, automated analyses.**

I. INTRODUCTION

A *software product line* is a set of software-intensive systems (program variants) that share common code artifacts [1]. The overall idea is to efficiently develop similar programs simultaneously, by systematically reusing development artifacts. Program variants are distinguished in terms of *features*, which are prominent or distinctive user-visible aspects, qualities, or characteristics of a software system [2]. Two variants may have several features in common and differ in other features. In particular, we focus on product-line engineering methods, in which program variants can be *generated* from a common implementation by specifying a selection of features. This includes implementation techniques, such as conditional compilation [3], plug-ins and frameworks, or advanced programming-language mechanisms with aspects [4], feature modules [5], [6], or delta modules [7].

In general, not all combinations of features are useful and result in meaningful program variants. For instance, there might be mutually exclusive features or features that require other features. A variability model specifies all valid *feature combinations* and thus the program variants that can be generated. A common form of variability models are feature models [2], [8], [9]. Feature models can have several representations, e.g., a propositional formula in which each feature belongs to a variable and the formula evaluates to true for all valid combinations [10].

We noticed that not all features in typical feature models are used to distinguish program variants. Some are only used to structure the model and selecting or eliminating them does not make any difference in the generated variant code. We denote such features as *abstract features*. Due to abstract features, the set of *feature combinations* and the set of *program variants* are not equivalent. Multiple valid feature combinations result in the same generated program variant.

There are numerous approaches to reason about valid feature combinations in feature models [9]. However, there are also many interesting questions, for which we want to reason about program variants. For example, for combinatoric testing [11], type checking [12]–[15], or verification [16], we want to select certain sets of program variants or need to reason about the relationship of features in program variants. Similarly, for reasoning about non-functional properties of program variants [17], abstract features are not relevant, but can increase measurement effort significantly. Finally, deleting an abstract feature from the feature model does not change possible program variants and may hence be considered as feature-model refactoring [18]. Contemporary automated analyses of feature models can be used to reason about valid combinations of features, but not to reason about program variants. When using automated reasoning nonetheless, the results are at least inaccurate or inefficient.

As a consequence, we distinguish two semantics: (a) the semantics of feature models as known from literature [19], describing the valid combinations of features and (b) the semantics of program families describing distinct program variants of a product line. According to our experience, both are needed and complementary to each other. However, in order to not redevelop all reasoning mechanisms again for the program-families semantics, we provide a mechanism to translate the program-families semantics in a way that existing reasoning mechanisms for the feature-model seman-

Thüm et al. SPLC11

Thüm et al. SPLC11

"A further helpful optimization is to replace abstract features by a definition in terms of concrete subfeatures. That is, variables representing an abstract feature are substituted by a disjunction of its subfeatures. We developed this technique in previous work, but in general, there is not always a definition for abstract features (especially in the presence of cross-tree constraints and abstract primitive features) [MIP]. Still, whenever possible, we use this substitution instead of the general pattern, and only fall back to the general pattern when required."

TL;DR: our reasoning was unsound for certain feature models with abstract features, but using slicing is sound

Remarks on Significance (Only 1 Day Old)

Tobias Heß in Home Office

"Hey Thomas,

I have good news!

I checked every commit in the history of Busybox, Fiasco, Soletta, uclibc, Toybox, and FinancialServices.

We can significantly simplify your classification algorithm, as 100 % of the changes are arbitrary edits:"

Remarks on Significance (Only 1 Day Old)

Tobias Heß in Home Office

"Hey Thomas,

I have good news!

I checked every commit in the history of Busybox, Fiasco, Soletta, uclibc, Toybox, and FinancialServices.

We can significantly simplify your classification algorithm, as 100 % of the changes are arbitrary edits:"

The New Algorithm

```
String classify(FM f, FM g) {
    return "Arbitrary Edit";
}
```

Dear reviewers, are you in the room?

Would you have accepted the paper
if you knew all this?

Conceptual Extensions

Classifying Edits to Non-Boolean FM

Dintzner et al. SoSyM17

Encoding More Than Two Versions

Nieke et al. GPCE18

Young et al. EMSE22

Encoding Evolution in Feature Model

Seidl et al. SPLC14

Conceptual Extensions

Classifying Edits to Non-Boolean FM

Dintzner et al. SoSyM17

Encoding More Than Two Versions

Nieke et al. GPCE18

Young et al. EMSE22

Encoding Evolution in Feature Model

Seidl et al. SPLC14

Saving Analysis Effort

Nieke et al. GPCE18

Thüm et al. VariVolution19

Classifying Edits to Feature Mappings

Bittner et al. FSE22

Classifying Edits to Product Lines

Dintzner et al. EMSE18

Edit Catalogs and Code Refactorings

(Partially) Safe Evolution

Borba et al. TCS12

Neves et al. JSS15

Sampaio et al. SPLC16

Nieke et al. SoSyM22

Edit Catalogs and Code Refactorings

(Partially) Safe Evolution

Borba et al. TCS12

Neves et al. JSS15

Sampaio et al. SPLC16

Nieke et al. SoSyM22

Refactoring of Variable Code

Schulze et al. AOSD13

Pietsch et al. ASE15, SPLC19

Liebig et al. ICSE15

Randomly Generated Models

first generator for random feature models

several months of computation time

models provided for download

Used As Is

Xue et al. WCRE11

Guo et al. ESWA12

Quinton et al. SPLC14

Arcaini et al. ICST15

Randomly Generated Models

first generator for random feature models

several months of computation time

models provided for download

Used As Is

Xue et al. WCRE11

Guo et al. ESWA12

Quinton et al. SPLC14

Arcaini et al. ICST15

Generator Extension

Segura et al. VaMoS12

Segura et al. ESWA14

Situation Today

"The March 1st 2019 version of the Gentoo distribution comprises 671,617 features spread across 36,197 feature models."
[Lienhardt et al. ICSE20]

Surprising Applications

Validating FM Composition

Acher et al. SLE09

Expressing FM Differences

Acher et al. CAiSE12

Validating FM Translations

Feichtinger et al. SPLC22

Surprising Applications

Validating FM Composition

Acher et al. SLE09

Expressing FM Differences

Acher et al. CAiSE12

Validating FM Translations

Feichtinger et al. SPLC22

Validating CNF Translations

Oster et al. SPLC10

Kuiter et al. ASE22

Feature Diagram Mutations

Arcaini et al. ICST15

Reuling et al. SPLC15

Conclusion & Future Work

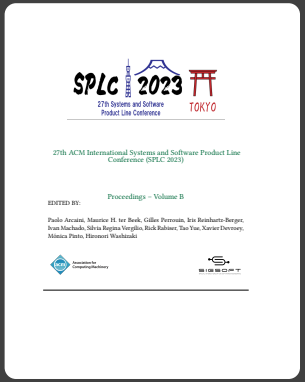
Conclusion

- Edits on feature models can be categorized in refactorings, generalizations, specializations and arbitrary edits
- Reasoning with propositional formula calculates the category of an edit
- Our optimizations scale this approach for feature models
  - with more than 1000 features
  - with more than 100 edits on a feature model

Future work

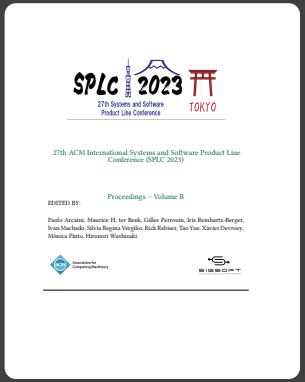
- Generalizing analysis to other variability models
- Finer distinctions of categories
- Compact visual representation of all added and removed products

Sorry Again for Not Reading Your Papers

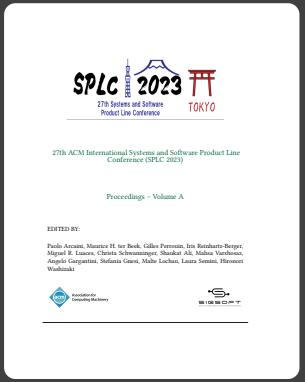


SPLC 2023
Proceedings — Volume A

Sorry Again for Not Reading Your Papers

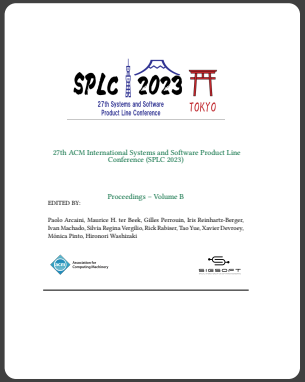


SPLC 2023
Proceedings — Volume A

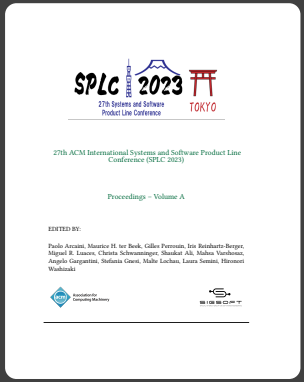


SPLC 2023
Proceedings — Volume B

Sorry Again for Not Reading Your Papers

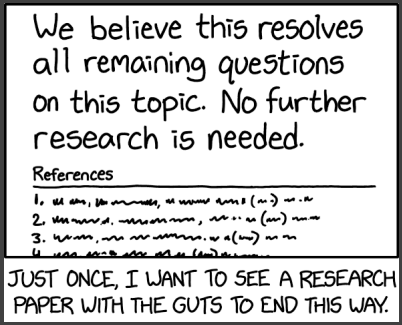


SPLC 2023
Proceedings — Volume A

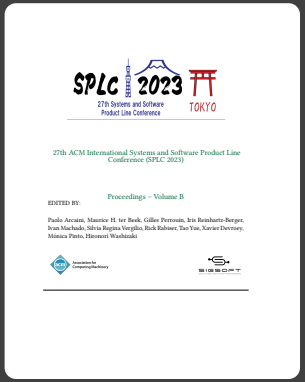


SPLC 2023
Proceedings — Volume B

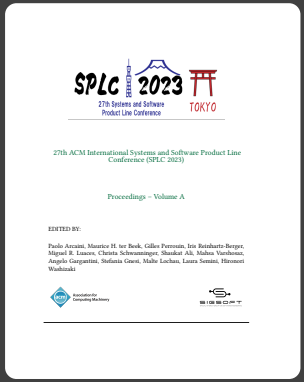
Let me end with this:



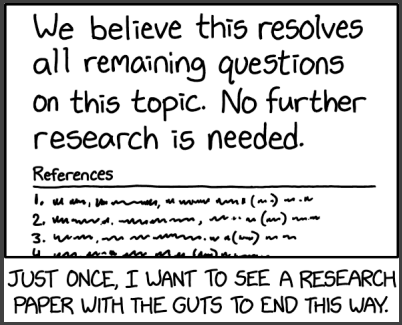
Thank You



SPLC 2023
Proceedings — Volume A



SPLC 2023
Proceedings — Volume B



. . . for your standing ovations ;o)

Reasoning About Edits to Feature Models (ICSE'09)