



Reasoning About Edits to Feature Models (ICSE'09)

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

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

Dan Ratory
Dept. of Computer Science
University of Texas at Austin
ratory@cs.utexas.edu

Christian Küster
School of Computer Science
University of Magdeburg
ckuester@ovgu.de

Abstract

Businesses represent the *variations and combinations* 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 are often subject to *edits*, i.e., changes to the feature model. An *edit* of a feature model via modifications as *refactorings*, specializations, generalizations, or arbitrary edits. We present an algorithm that can reason about edits to feature models. It helps to determine how the program membership in an SPL has changed. Our algorithm takes two feature models as input and determines whether the sets of features in both models are not necessarily the same and automatically computes the change classification. Our algorithm is able to handle edits that add or delete features and efficiently classifies edits to very 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 a domain share a common set of features which are abstractions relevant to stakeholders and are typically increments in program functionality. Every program of an SPL is a variation of a common feature model. A single feature model could have millions of distinct programs. A *feature model* completely defines all features in an SPL and their configurations, it is basically an AND/OR graph with constraints.

SPLs and their feature models evolve over time. Even small edits to a feature model, e.g., adding one feature to one branch to another, can unintentionally change the set of legal feature combinations. For example, edits may produce invalid configurations that do not correspond to any of the sets 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, Camurati et al. defined operations to specialize feature models [13, 14, 25], where a feature model X is specialized to a feature model Y . This means that Y is a subset of that in X . Ahola et al. discussed operations called *refactorings* that maintain the set of products or add new products to a feature model [1]. This means that the new products are added to a *generalization* of a feature model.

In both approaches, a set of sound operations is used to edit feature models. In our work, we extend these operations to handle edits that add or delete features. We also introduce a notion of *edit classification* that distinguishes between edits that reflect a feature model's evolution (refactorings, generalizations, or specializations). However, a tool that allows only sound operations to be performed is not sufficient to answer questions such as, 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 generalization operation does not necessarily create a sequence of edits (if it exists). If a required operation is missing, designers would be out of luck, as the desired transformation cannot be performed without the operation provided by the system.

In other work, Sim et al. used first-order logic to determine if two feature models are equivalent, i.e., both models have the same set of programs that can be built [30]. They also used higher-order logic to determine whether a feature model is a specialization of another [29]. Both approaches are not applicable to model edits that add or delete features. Further, we are aware of performance studies that demonstrate that reasoning with first-order logic approaches scale to large feature models efficiently [20].

In this paper, we present and evaluate an algorithm to determine the effect of edits to feature models (i.e., refactorings, specializations/generalizations of these) using satisfiability solvers. We assess the limitations of

Let Us Do A Dry Run

Reasoning about Edits to Feature Models

Thomas Thüm¹ Dan Suciu² Christian Kästner³

¹School of Computer Science
University of Magdeburg
thuemsef.org/research

²Dept. of Computer Science
University of Texas at Austin
dsuciu@cs.utexas.edu

³School of Computer Science
University of Magdeburg
ckaestne@ovgu.de

Abstract

Feature engines the variability and compatibility 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 the SPL. Given an initial feature model and edits to it, we want to determine whether the resulting feature model is valid or not. We also want to determine the evolution of a feature model via modifications as refactoring, generalization, specialization, and addition/deletion of features. In this paper, we present an algorithm to reason about feature model edits to help designers determine how the program membership of an SPL changes under edits. Our algorithm takes two feature models as input (before and after edit execution), where the set of features in both models are not necessarily the same, and designs edits to make them compatible. Our algorithm is able to give examples of added or deleted products and efficiently handles edits to even large models that have thousands of features.

1 Introduction

Software product line (SPL) engineering is a paradigm for reuse [1, 2, 3]. From a feature model, different programs of a domain can be assembled. Programs in an SPL are distinguished by *features*, which are discrete elements that are combined to realize different configurations in program functionality. Every program of an SPL is represented by a unique combination of features, and an SPL is a collection of such 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 location to another, may result in the removal of legal feature combinations. For example, edits may preclude the creation of programs that were previously built or enlarge the set of programs that are now valid. Understanding the impact of feature model edits is known to be important in

determining 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 need.

In prior work, Camenisch et al. defined operations to specialize feature models [13, 14, 25], and to refine a feature model X into a new feature model X' . They also introduced the notion of a *feature model edit* [14], which is a subset of that of Alten et al., discussed operations called *rule insertion* that either add a new rule to a product or add a new rule to a feature model [15]. We call this class of edits *refactorings*. When new products are added, a generalization of a feature model is often required [16]. We introduce a new operation to refine a feature model to determine whether a model is a specialization or refactoring or generalization of another. To determine whether edits are valid, we propose an algorithm that guarantees that all edits be expressed as a sequence of predefined operations (refactorings, generalizations, or specializations). However, the complexity of this algorithm is high, and a performed model would have the feel of a hard-to-use feature editor, e.g., how one changes a feature model X into a larger feature model X' by adding a new feature f and changing other features. A path planner [17] would be needed to automatically compute a sequence of edits to edit. If a sequence operation is used, then the problem of determining whether a sequence of edits to a feature model would be precluded by the edits. That approach seems too slow for practical use.

There are two approaches to model edits: first-order logic to determine if two feature models are equivalent, i.e., both models have the same set of products [17], Janusz and Kosuszko proposed a logic to determine if a feature model is a generalization or specialization of another [28]. Both approaches work if both models have the same set of features, their results are not applicable to edits that add or remove features. Further, we are unaware of performance studies that demonstrate first-order and higher-order logic approaches scale to large feature models.

In this paper, we present and evaluate an algorithm to determine the relationship between two feature models (i.e., generalization, specialization, and equivalence) of feature models. We evaluate the limitations of

Let Us Do A Dry Run

Reasoning about Edits to Feature Models

Thomas Thüm
School of Computer Science
University of Magdeburg
thuemset.org/qu.de

Dan Berry
Dept. of Computer Science
University of Texas at Austin
batory@cs.utexas.edu

Christian Kämer
School of Computer Science
University of Magdeburg
ckaem@se.cs.uni-magdeburg.de

Abstract

Feature engines the variability and combinatorics among programs in a software product line (SPL). A feature model defines the valid combinations of features, where each combination corresponds to a unique program. The edits to a feature model can change the set of legal feature models and their feature models evolve over time. We describe the evolution of a feature model via modifications as refactoring, specifying, and deleting operations. We present an algorithmic approach to reason about feature model edits to help designers determine how the program membership of an SPL changes over time. We also show how edits to feature models are input (before and after edit version), where the set of features in both models are not necessarily the same, and distinguish between edits to feature models. Our algorithm is able to give examples of added or deleted products in an efficiently classified edit to even large models that have thousands of features.

1 Introduction

Software product line (SPL) engineering is a paradigm that requires reuse [2, 11]. From a feature perspective, different programs of a domain can be assembled. Programs of an SPL are distinguished by *features*, which are discrete elements that are combined to build programs and contribute to program functionality. Every program of an SPL is represented by a unique combination of features, and an SPL is a collection of such programs. A feature model compactly defines all features in an SPL and their model compatibility; 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 location to another, can change the set of legal feature combinations. For example, edits may precede the creation of programs that reuse previous, built or ongoing, feature configurations. Underlying the evolution, the impact of feature model edits is known to be impacted in

deterministic manner. 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 need.

In prior work, Camello et al. defined operations to specialize feature models [13, 14, 25], where a feature model M is refined into a new feature model M' via an operation δ . In [3] it is shown that if δ is a refinement of M , then M' is a subset of M . Alves et al. discussed operations called *rule insertion* that add rules to a feature model [1]. We call this kind of operations *specifications*. When new products are added, a generalization of a feature model is often required [17]. We call this kind of operations *generalizations*. Both approaches require a feature model and a feature model to determine whether a model is a specialization or a generalization of another. To determine whether a feature model is a generalization of another we prove that all edits are expressed as a sequence of predefined operations (refactorings, generalizations, or specifications).

However, the complexity of feature models that are performed would have the feel of a hard-to-use structure editor, e.g., how one changes a feature model M into a larger feature model M' is not always clear, and how many edits are needed.

A path planning [17] would be needed to automatically compute a sequence of edits. If a sequence operation is provided, then the problem is reduced to how to find a valid sequence of edits. We propose a sequence of edits to a feature model that would be produced by the editor. That approach uses tree automata [18] to validate edits. We also propose to use first-order logic to determine if two feature models are equivalent, i.e., both models have the same set of products [17]. Janusz and Kosusz proposed a logic to validate edits to feature models against a specification of another [20]. Both approaches work if both models have the same set of features. Their results do not apply to edits that add or remove features. Further, we are aware of performance studies that demonstrate first-order and higher-order logic approaches scale to large feature models [19].

In this paper, we present and evaluate an algorithm to determine the relationship between two feature models (i.e., generalization, specialization) and the use of feature satisfiability solvers. We overcome the limitations of

ICSE'09, May 14–24, 2009, Vancouver, Canada
978-1-4244-3825-3/09 \$25.00 © 2009 IEEE

258

Thüm et al. ICSE09



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

Proceedings – Volume B

EDITED BY:

Paolo Arcaini, Maurice H. ter Beek, Gilles Perronin, Iris Reinhardt-Berger,
Ivan Machado, Silvia Regina Vergilio, Rick Balster, Tao Yue, Xavier Deverny,
Mónica Pinto, Hiroshi Washizaki



Association for
Computing Machinery



SPLC

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
thuem@cs.uni-magdeburg.de

Dan Berry
Dept. of Computer Science
University of Texas at Austin
berry@cs.utexas.edu

Christian Kästner
School of Computer Science
University of Magdeburg
ckaestner@cs.uni-magdeburg.de

Abstract

Feature models express the variability and combinatorics among programs in a software product line (SPL). A feature model defines the valid combinations of features, where each combination corresponds to a unique program. The *memberships* of a feature model are sets of programs, and these feature models overlap over time. We describe the evolution of a feature model via modifications as refactoring, specifying, and generalizing operations. We present an algorithm to reason about feature model edits to help designers determine how the program membership of an SPL changes over time. Our algorithm takes feature models as input (before and after edit version), where the set of features in both models are not necessarily the same, and designs a sequence of operations to transform one model into the other. We evaluate our algorithm by giving it edits to solve examples of added or deleted products, and it efficiently classifies edits to even large models that have thousands of features.

1 Introduction

Software product line (SPL) engineering is a paradigm that allows reuse of code [2, 11]. From a feature model, programs of a domain can be assembled. Programs are distinguished by *features*, which are discrete elements that are combined to build programs and contribute to program functionality. Every program of an SPL is represented by a unique combination of features, and an SPL is a collection of feature models. 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 location to another, may change the set of valid legal feature combinations. For example, edits may preclude the creation of programs that were previously built or enlarge the set of programs that are now valid. Understanding the impact of feature model edits is known to be important in

determining 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 research problem.

In prior work, Camenisch et al. defined operations to specialize feature models [13, 14, 25], and a feature model \mathcal{F} is a specialization of another feature model \mathcal{F}' if \mathcal{F} is a subset of \mathcal{F}' . In fact, it is not always the case that new products are added as generalizations of a feature model. For example, a feature model \mathcal{F} may be refined by adding new products or adding new features to \mathcal{F} . We call this type of edits *refactorings*. Refactorings do not change the set of valid programs, but they change the way feature models are used to determine whether a model is a specialization or a generalization of another. To determine whether a model is a specialization or a generalization of another, we propose an algorithm that all edits are expressed as a sequence of predefined operations (refactorings, generalizations, or specializations).

However, this approach is not always feasible. A feature model would have the feel of a hard-to-use structure editor, e.g., how one changes a feature model \mathcal{F} into a larger feature model \mathcal{F}' is not always clear, and a sequence of edits that change \mathcal{F} into \mathcal{F}' would not always be unique. A path planner [17] would be needed to automatically compute a sequence of edits to edit. If a sequence operation is applied to a feature model, it is not always clear what change to a feature model would be produced by the edit. That approach seems too complex for practical use.

Instead, we propose to use first-order logic to determine whether two feature models are equivalent, i.e., both models have the same set of products [17]. Janusz and Kosuszko introduced a logic for reasoning about feature models [26] to specify evolution of another [28]. Both approaches work if both models have the same set of features, their results do not apply to edits that change the set of features. Further, we are unaware of performance studies that demonstrate first-order and higher-order logic approaches scale to large feature models.

In this paper, we present and evaluate an algorithm to determine the relationship between two feature models (i.e., generalizations, specializations, and refactorings) of feature models using satisfiability solvers. We evaluate the limitations of



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

Proceedings – Volume B

EDITED BY:

Paolo Arcaini, Maurice H. ter Beek, Gilles Perrousin, Iris Reinhardt-Berger, Ivan Machado, Silvia Regina Vergilio, Rick Balster, Tao Yue, Xavier Deverny, Mónica Pinto, Hiromi Washizaki



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

Proceedings – Volume A

EDITED BY:

Paolo Arcaini, Maurice H. ter Beek, Gilles Perrousin, Iris Reinhardt-Berger, Miguel R. Lucio, Christa Schwanninger, Shankat Ak, Maha Varshosaz, Angelo Gargantini, Stefania Gnesi, Malte Lohaus, Laura Semini, Hiromi Washizaki



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

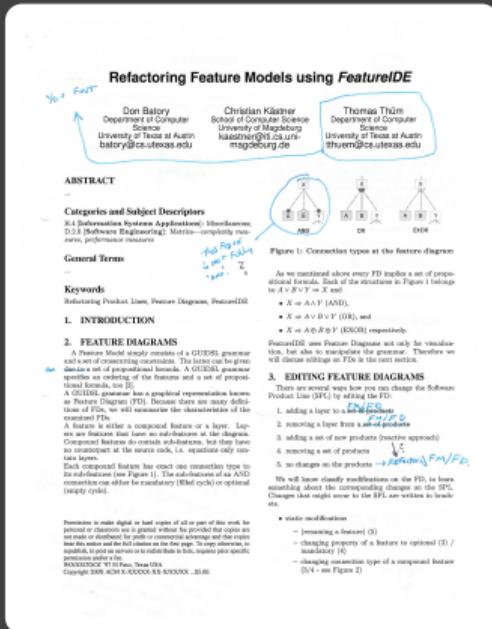
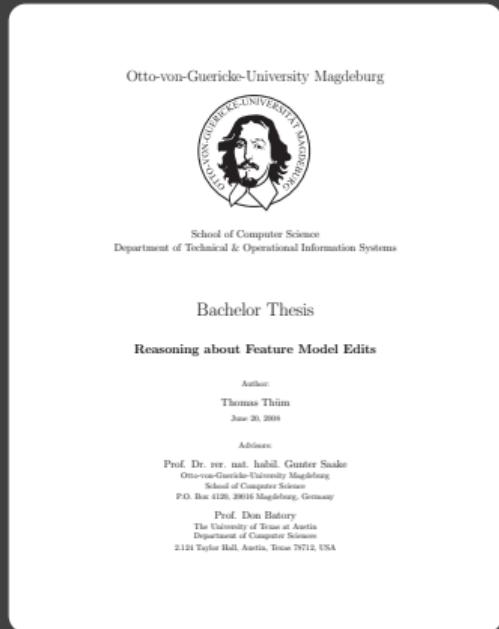
Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Otto-von-Guericke-University Magdeburg
Sachsenplatz 12
P.O. Box 4120, 39014 Magdeburg, Germany

Prof. Dan Boneh
The University of Texas at Austin
Department of Computer Sciences
2124 Taylor Hall, Austin, Texas 78712, USA

submitted June 2008

My Bachelor's Thesis



submitted June 2008

started to write a paper

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

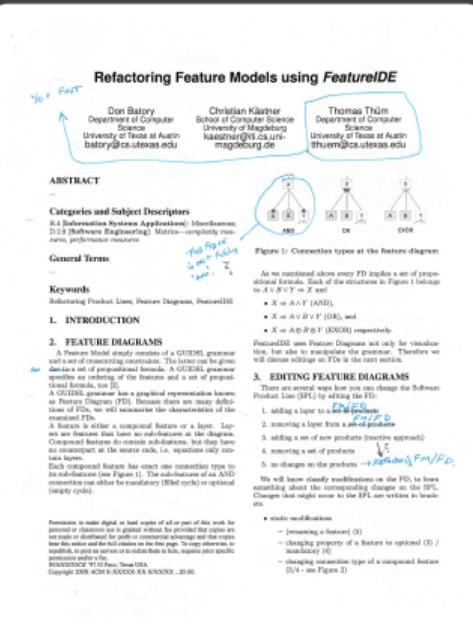
卷之三

4.000000 4.000000

Prof. Dr. rer. nat. habil. Gunter Saake
Ottovon-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
2124 Taylor Hall, Austin, Texas 78712, USA

submitted June 2008



Reasoning about Edits to Feature Models

Thomas Thüm
School of Computer Science
University of Magdeburg
t.thuem@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
cikastne@ovgu.de

Abstract

feature express the variables and communicating progress in a software product line (SPL). A feature often defines the valid configurations of features, where each combination corresponds to a program in an SPL. SPLs are typically used to manage the evolution over time. We propose the notion of a feature model as a classification system as regular specializations, generalizations, or arithmetic axioms. We present an algorithm to reason about feature model axioms. Our designers determine here about the program membership if the SPL has changed. Our algorithm takes two feature models as input (before and after edit session), where the former is the base model and the latter is the target model. It automatically computes the classification changes. Our system is able to give examples of added or deleted products and efficiently classifies features in even large models that thousands of features.

Introduction

Software product line (SPL) engineering is a paradigm for reuse [2], [3], [20]. In SPL engineering, multiple programs of a domain can be assembled. Programs in SPLs are distinguished by *features*, which are domain requirements relevant to stakeholders and are typically incarnated in program functionality. Every program in an SPL is represented by a unique combination of features, and an SPL can have millions of distinct programs. A *feature composition* defines all instances in an SPL, and their combinations; it is basically an AND-OR graph with constraints.

changes to a feature model, like moving a feature from branch to another, can unintentionally change the set of feature combinations. For example, edits may produce variations of packages that were previously built or enlarge

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

determine manually. Tool support to provide feedback to engineers to explain how changes to a feature model have affected the program membership of an SPL is a fundamental requirement.

In order words, Cremaditi et al. defined operations as

In general, while it is common to use minimum spanning trees to realize feature models [13, 14, 25], where a feature model F is a **specialization** of a feature model F' if the set of products $\text{prod}(F)$ is a subset of that in F , Alves et al. discussed operations called **reflections** that maintain the set of products.

a refactoring that maintains the set of products in its SPL [1]. We call this latter case (where no products are added) a **generalization** of a feature set. In both approaches, a set of sound operations is used to create 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

5-24, 2009, Vancouver, Canada
ISBN 978-0-7803-1500-0 © 2009 IEEE

started to write a paper

submitted August 2008

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



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

clear to you :)

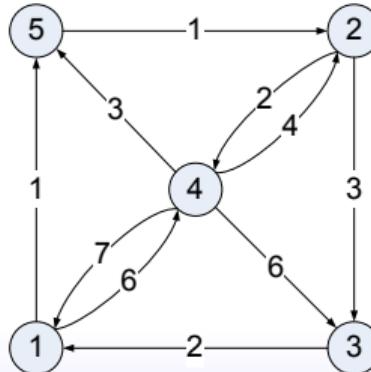
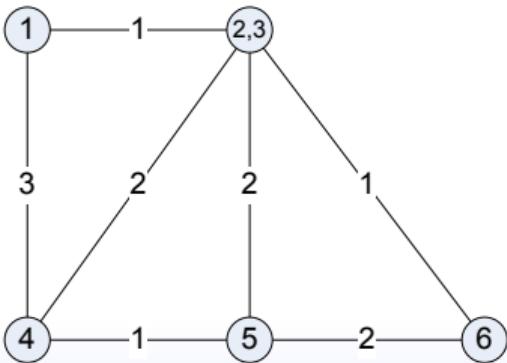
Software product line (SPL)

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



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

Author(s): Lopez-Herrejon and Batory
Organization(s): University of Texas
Austin, USA
(Email contact) lherrejon@cs.utexas.edu

Abstract: This paper presents a standard problem for evaluating product-line methodologies. It is based on a well-known problem in the field of feature modeling, namely, determining the cost of a configuration. The basic idea is to present a problem that is representative of the complexity of real-life problems. The problem is designed to be simple enough so that it can be solved by hand, yet complex enough to require computer support to obtain feasible solutions.

1. Introduction

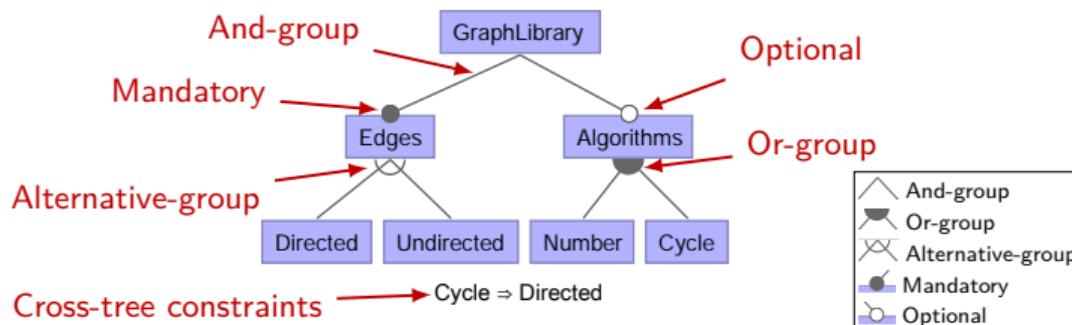
A product line is a family of related software applications. A product line architecture is a design for a product line that defines the commonalities and variations among the products in the line. The goal of a product-line methodology is to support the reuse of commonalities and to reduce the cost of variation. The basic idea is to reuse commonalities across the products in the line, yet still be able to quickly and easily change individual components without affecting other family members from them. This paper presents a standard problem for evaluating product-line methodologies. It is based on a well-known problem in the field of feature modeling, namely, determining the cost of a configuration. The basic idea is to present a problem that is representative of the complexity of real-life problems. The problem is designed to be simple enough so that it can be solved by hand, yet complex enough to require computer support to obtain feasible solutions.

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:



also clear to you :)

but this is our running example

- ▶ 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



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

A screenshot of a journal article page from the Journal of Systems & Software (JSS). The title is "Comparing the intensity of variability changes in software product line evolutions". The authors are Christian Kröher and Lin-Geling Klaas. The journal is published by Elsevier, Volume 233, ISSN 0888-613X, ISSN 1084-8031, DOI 10.1016/j.jss.2023.112000, Available online 10 January 2023. The abstract discusses the evolution of feature models in software product lines, focusing on the intensity of variability changes. It compares different metrics and tools for tracking these changes over time.

Kröher et al. JSS23

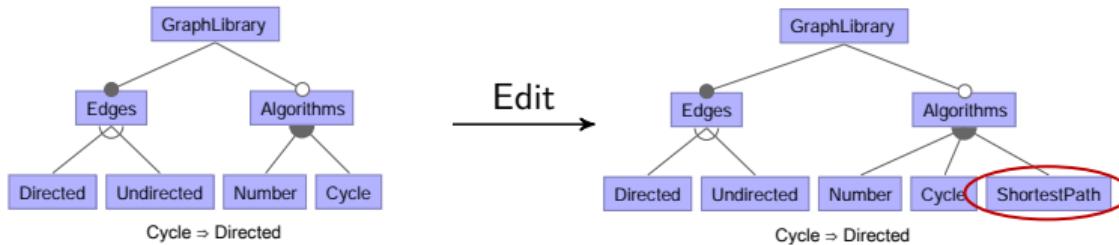
4 % of commits in Linux kernel affect feature model

1 % solely change the feature model

What Is Our Contribution?

Generalization

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



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



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

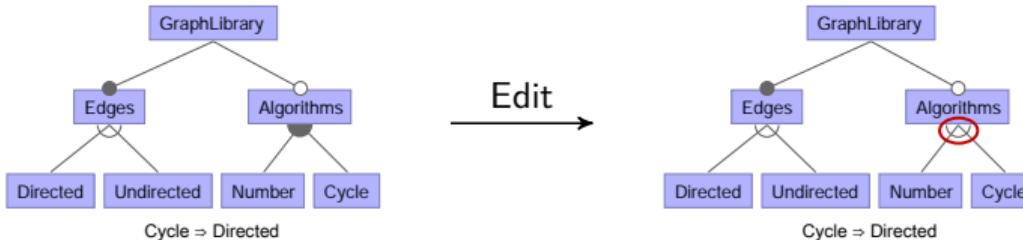
Alves et al. GPCE06
(edit patterns)

motivation for our work



Specialization

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



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

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

Reasoning About Feature Model Specializations

Reinhard Pichler¹, Stefan Wolter², and Michael Beiner²

¹ University of Bayreuth, Germany

² University of Applied Sciences Münster, Germany

Abstract. Feature modeling has been proposed to capture the hierarchical and modular nature of software systems. In particular, feature models have been used to support reuse by specifying how existing features can be composed into new ones. Many configurations in a system that do not necessarily correspond to valid products are considered as invalid. This is done to prevent invalid configurations from being selected by automated tools. In this paper we propose a formal framework for reasoning about edits to feature models. We make this precise by specifying how a feature model is modified by an edit. We then show how edits can be checked for validity. Finally, we show how edits can be checked for consistency. Consistency is important to ensure that the resulting feature model is still valid. We also show how edits can be checked for compatibility. Compatibility is important to ensure that the resulting feature model is still valid and can be composed with other feature models.

Key words: Feature model, product line, reasoning, consistency, compatibility

1 Introduction

Feature modeling is widely adopted to capture and manage the common and variable features in a product line. Feature models have been used to support reuse by specifying how existing features can be composed into new ones. Many configurations in a system that do not necessarily correspond to valid products are considered as invalid. This is done to prevent invalid configurations from being selected by automated tools.

In this paper we propose a formal framework for reasoning about edits to feature models. We make this precise by specifying how a feature model is modified by an edit. We then show how edits can be checked for validity. Finally, we show how edits can be checked for consistency. Consistency is important to ensure that the resulting feature model is still valid. We also show how edits can be checked for compatibility. Compatibility is important to ensure that the resulting feature model is still valid and can be composed with other feature models.

Edits to feature models have certain features. Amongst others, feature configuration

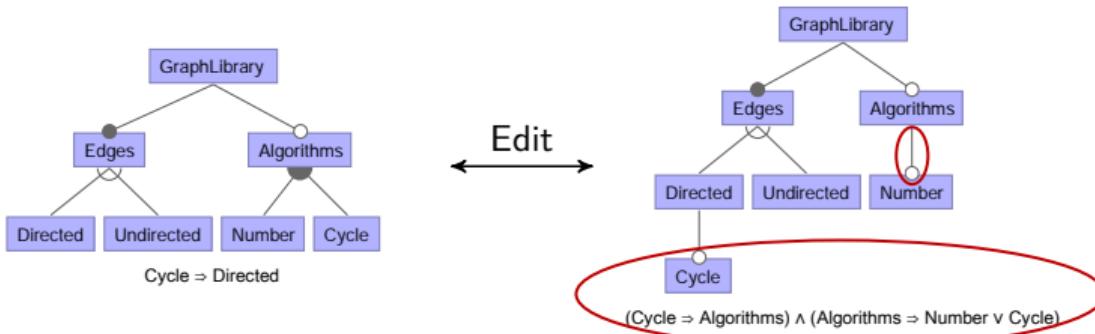
Czarnecki et al.
SPIP05

(staged
configuration)



Refactoring

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



$$\begin{aligned} &\{G, E, D\}, \{G, E, U\}, \\ &\{G, E, D, A, N\}, \{G, E, D, A, C\}, \\ &\{G, E, U, A, N\}, \{G, E, D, A, N, C\} \end{aligned}$$

=

$$\begin{aligned} &\{G, E, D\}, \{G, E, U\}, \\ &\{G, E, D, A, N\}, \{G, E, D, A, C\}, \\ &\{G, E, U, A, N\}, \{G, E, D, A, N, C\} \end{aligned}$$

Refactoring Product Lines

Abstract

Introduction

Related Work

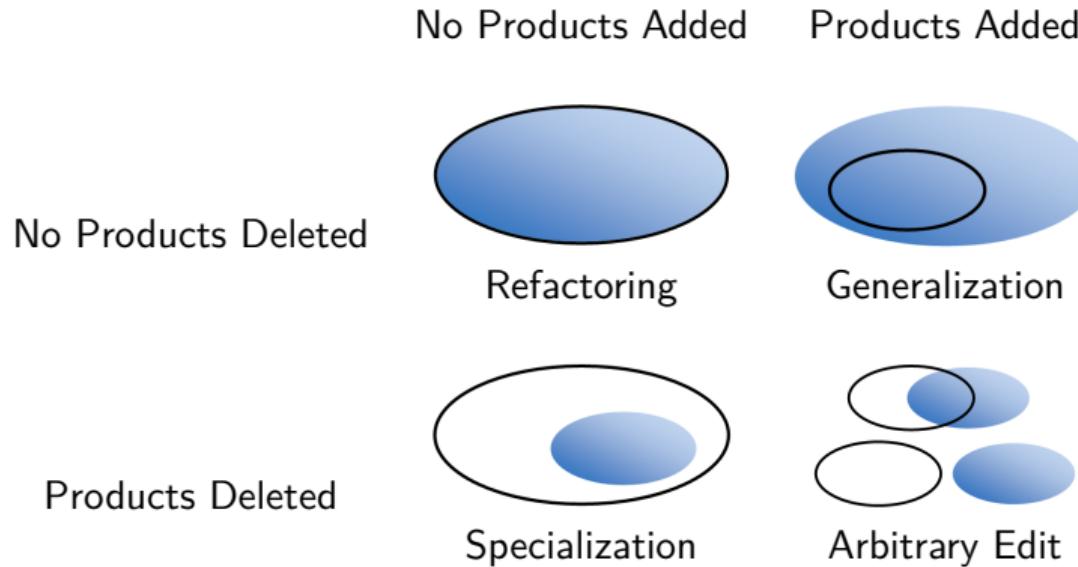
Conclusion

Alves et al. GPCE06
(edit patterns)

motivation for our
work



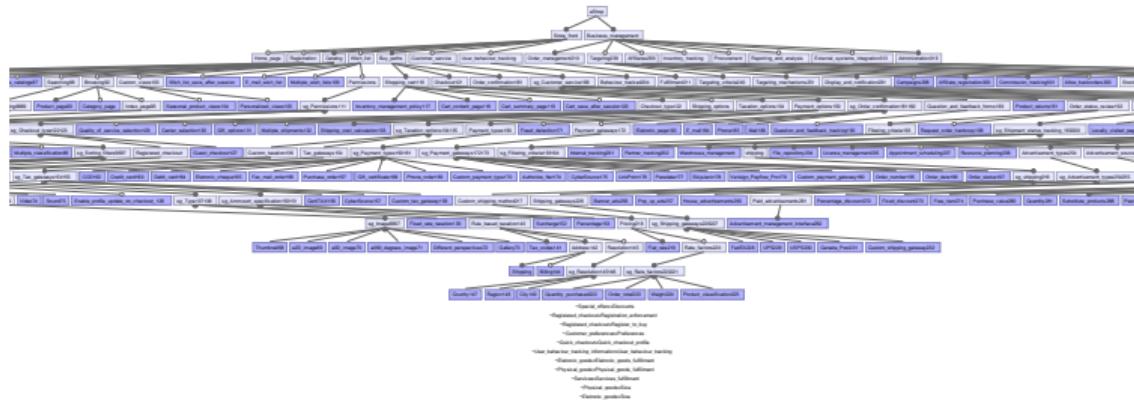
Categories of Edits





Our Goal

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



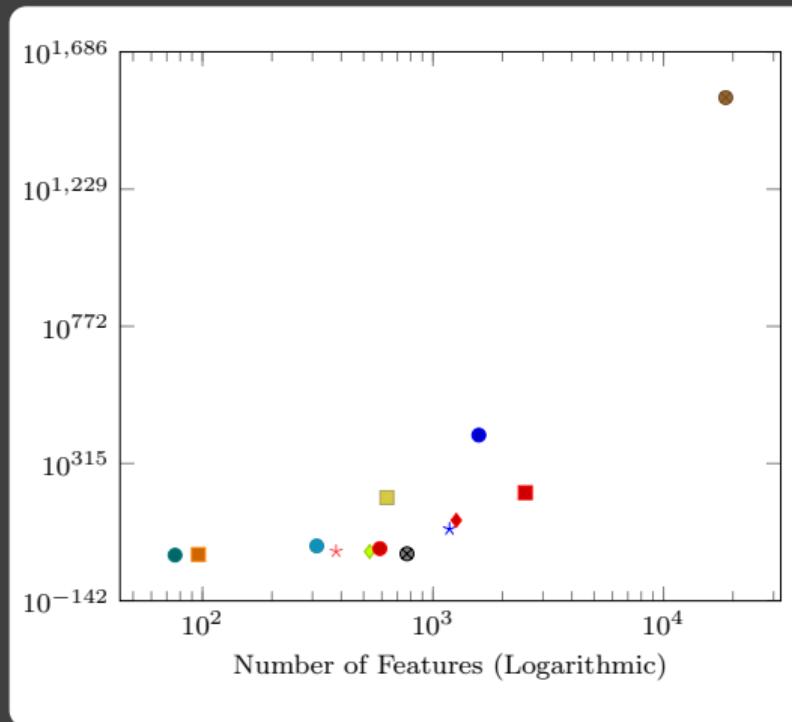
Mendonca, University of Waterloo

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



$$\begin{aligned} & \{G, E, D\}, \{G, E, U\}, \\ & \{G, E, D, A, N\}, \{G, E, D, A, C\}, \\ & \{G, E, U, A, N\}, \{G, E, D, A, N, C\} \end{aligned}$$

?

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

Enumeration does not scale!

Software License Agreement <hr/> Software, Application No. 45.ET. Software am Standard Configuration System	
Name: John Doe Address: Private residence - 123 Main Street, Pleasanton, San Mateo County	
Serial No.: 1234567890	
Version No.: 1.0	
Date: 12/31/2023	
<p>Software License Agreement</p> <p>This Software License Agreement ("Agreement") is made and entered into by and between Software, Application No. 45.ET, located at 123 Main Street, Pleasanton, San Mateo County, California, USA ("Licensor"), and John Doe, located at Private residence - 123 Main Street, Pleasanton, San Mateo County ("Licensee").</p> <p>Licensor grants to Licensee a non-exclusive, non-transferable license to use the Software, which is defined as the computer program and associated documentation provided by Licensor, in its standard configuration system, for the purpose of managing families of products. The Software is intended for use in conjunction with a personal computer and other hardware and software components.</p> <p>The Software is a complex system consisting of multiple modules and components. The Software is designed to manage families of products, including but not limited to, tracking product information, managing inventory, generating reports, and providing analytical insights. The Software is intended for use in a business environment, such as a manufacturing or distribution company, where it can be used to manage various aspects of the product lifecycle.</p> <p>Licensee agrees to use the Software in accordance with the terms and conditions of this Agreement. Licensee shall not reverse engineer, decompile, or otherwise attempt to gain access to the source code or underlying algorithms of the Software. Licensee shall not sublicense, resell, or otherwise transfer the Software without the prior written consent of Licensor.</p> <p>Licensor reserves the right to make changes to the Software from time to time, subject to notice to Licensee. Licensee shall have the right to terminate this Agreement if Licensor makes changes to the Software that significantly alter its functionality or performance.</p> <p>Both parties shall keep all information related to the Software confidential and shall not disclose such information to any third party without the prior written consent of the other party. This Agreement does not grant any rights to either party to use the Software for any purpose other than the purpose set forth in this Agreement.</p> <p>Software, Application No. 45.ET, is a registered trademark of Software, Application No. 45.ET, Inc.</p> <p>John Doe, 123 Main Street, Pleasanton, San Mateo County, California, USA</p>	

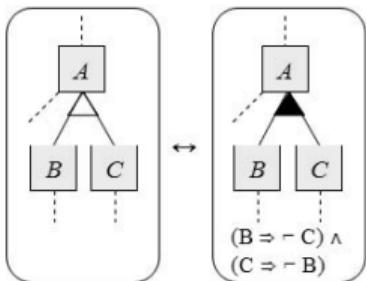
Sundermann et al.
EMSE23

(size of configuration spaces)

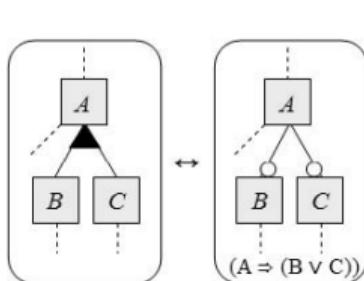


2. Sound Operations

- ▶ 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*

This screenshot shows a page from a technical report titled "Refactoring Product Lines". The page contains several sections of text, tables, and figures. At the top, there are four small tables under the heading "Refactoring Product Lines" showing details like "Number of Edits", "Number of Lines", "Number of Features", and "Number of Products". Below these are sections titled "Introduction", "Motivation", "Edit Patterns", "Implementation", "Evaluation", and "Conclusion". The "Edit Patterns" section contains two diagrams, one for "Replace Alternative Refactoring" and one for "Replace Or Refactoring", which are identical to the ones shown in the main slide.

Alves et al. GPCE06
(edit patterns)

motivation for our
work



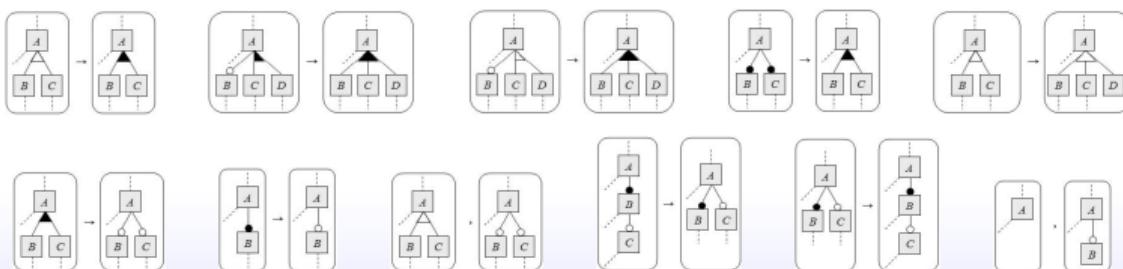
2. Sound Operations

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



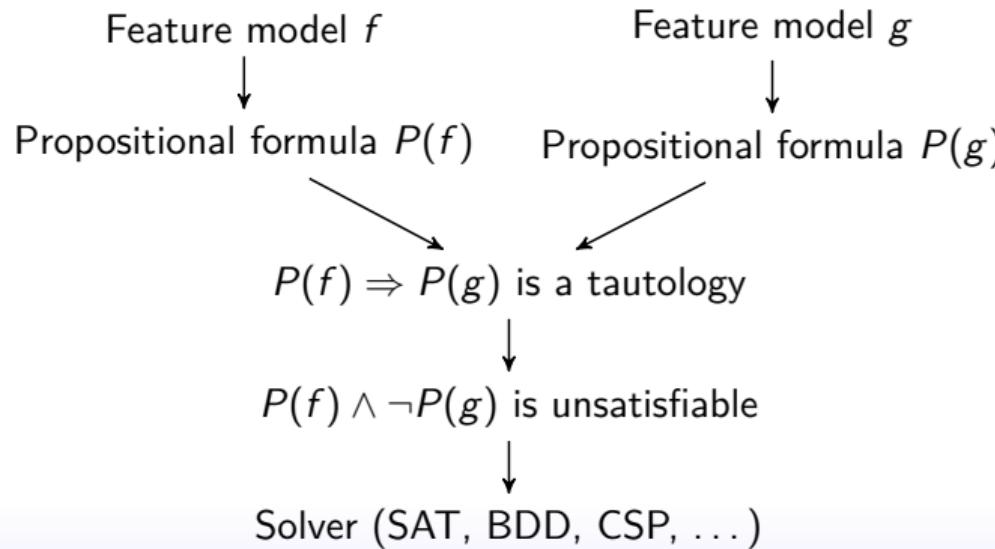
Alves et al. GPCE06
(edit patterns)

motivation for our work



3. Reasoning using Propositional Formulas

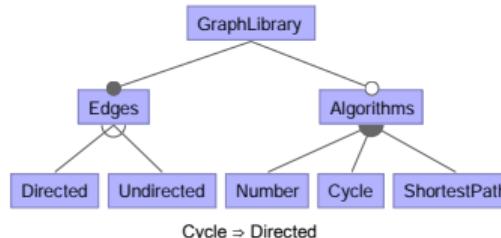
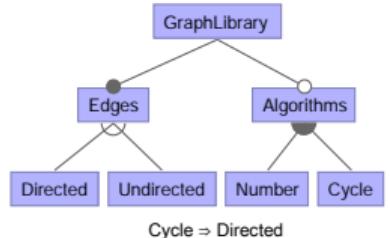
- ▶ Proposed by Sun et al. in 2005, Janota and Kiniri in 2007
- ▶ Are all products from feature model f available in g ?





3. Reasoning using Propositional Formulas

Standard translation into a propositional formula (Batory 05)



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

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

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



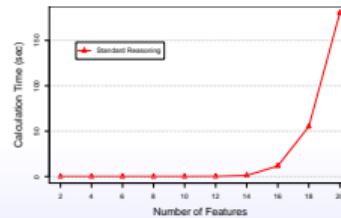
3. Reasoning using Propositional Formulas

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!





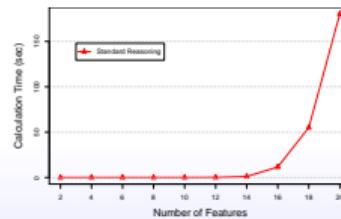
3. Reasoning using Propositional Formulas

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 . . .



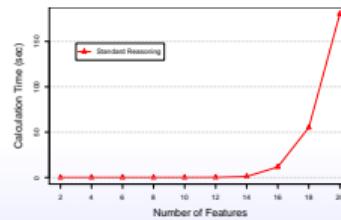
3. Reasoning using Propositional Formulas

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?"

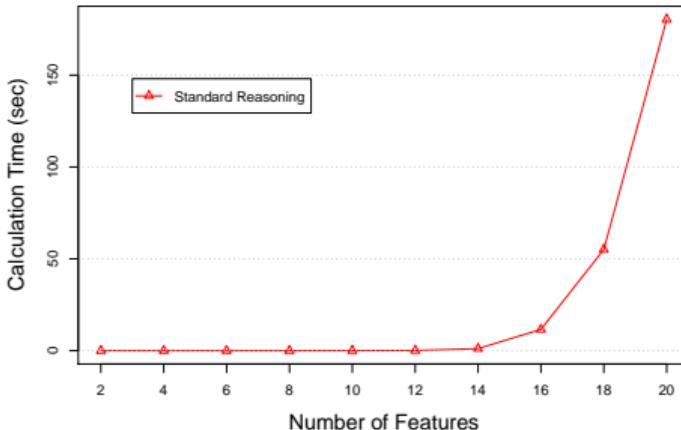
"I could work as a
programmer in
industry . . ."

A close-up shot of a man with dark, curly hair and a surprised, shouting expression. He has wide eyes and his mouth is wide open with his tongue slightly out. He is wearing a light gray, short-sleeved t-shirt. The background is blurred, showing what appears to be an indoor setting with warm lighting.

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

SAT Encoding of the *de-Munk* Constraint
A Case Study on Outgassing Boundary Variables
David Salomon-Mendelsohn, Yael Shabtai, Michael Zilberman
Department of Electrical Engineering, Tel-Aviv University,
69978 Tel-Aviv, Israel
E-mail: dsalomon@eng.tau.ac.il, yaelshabtai@tau.ac.il, mzilberman@tau.ac.il

Abstract. In constrained optimization problems, it is often required to encode boundary conditions as constraints. This paper presents a case study on the encoding of boundary variables in the context of the *de-Munk* constraint, which is a boundary condition that limits the number of transitions between two states. The paper shows how to encode this constraint using the SAT (Satisfiability) problem, and provides a detailed analysis of the resulting SAT instance. The results show that the SAT instance is highly structured and can be solved efficiently using standard SAT solvers.

Keywords: Constrained optimization, boundary conditions, SAT, de-Munk constraint.

1. Introduction

In discrete optimization problems, it is often required to encode boundary conditions as constraints. This paper presents a case study on the encoding of boundary variables in the context of the *de-Munk* constraint, which is a boundary condition that limits the number of transitions between two states. The paper shows how to encode this constraint using the SAT (Satisfiability) problem, and provides a detailed analysis of the resulting SAT instance. The results show that the SAT instance is highly structured and can be solved efficiently using standard SAT solvers.

Bittner et al. SEFM19

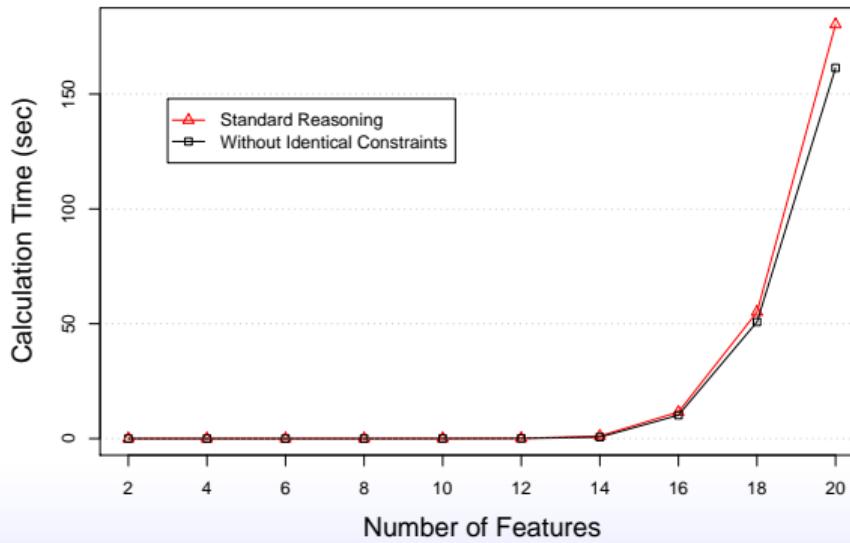
Kuiter et al. ASE22
translation to CNF is
not always easy



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



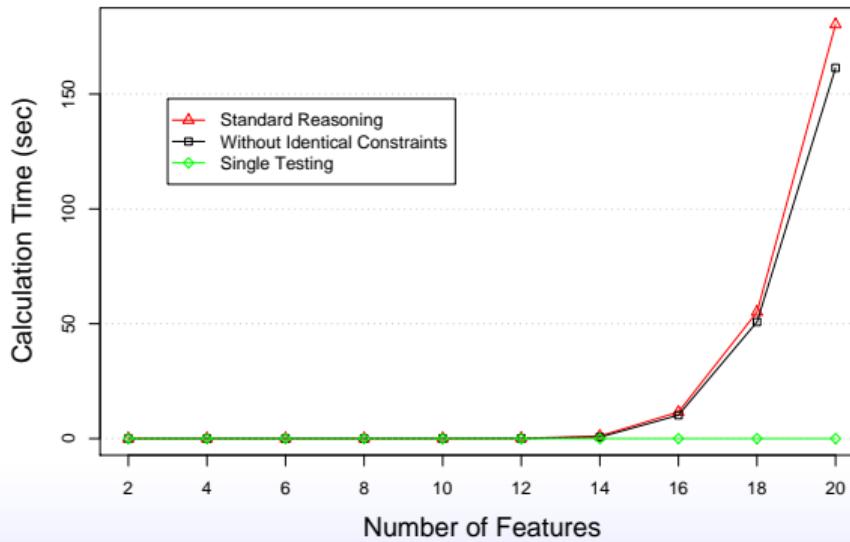
details in the paper



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



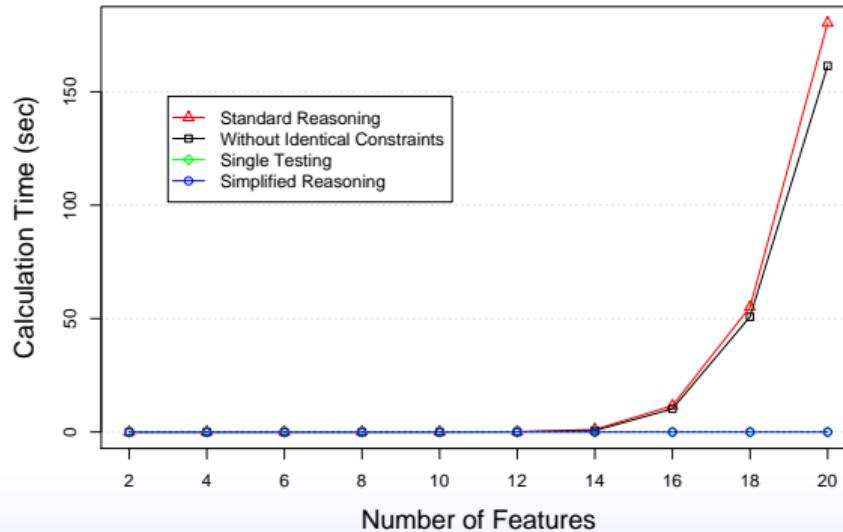
the core contribution!



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

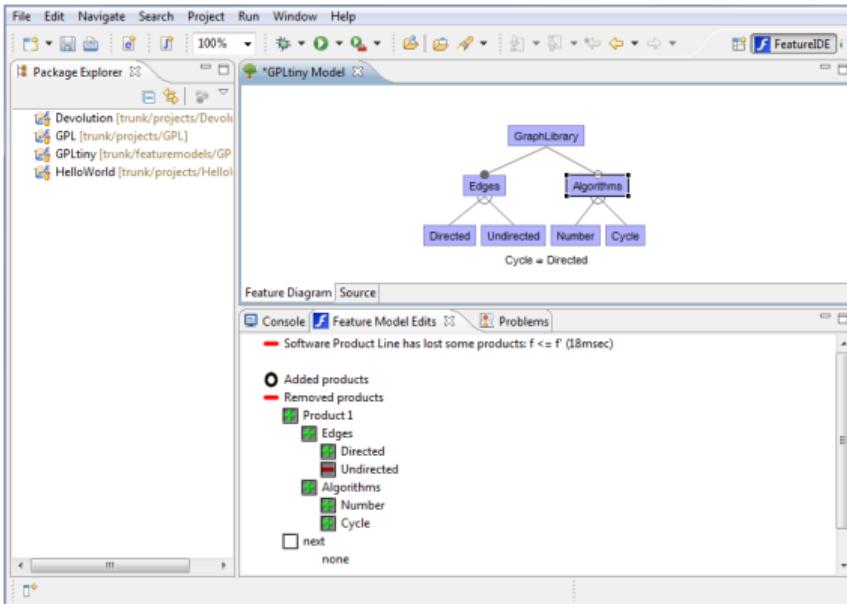
details in the paper

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



Implemented in FeatureIDE



sorry for increasing
YOUR environmental
footprint

Formal tool demonstration tomorrow at 4:30pm

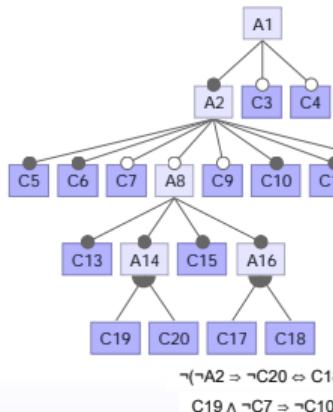
Available open source at <http://www.fosd.de/featureide>

Does It Scale?



Does Our Approach Scale?

- ▶ Do our optimizations allow reasoning about large feature models within reasonable time?
- ▶ Unable to acquire 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

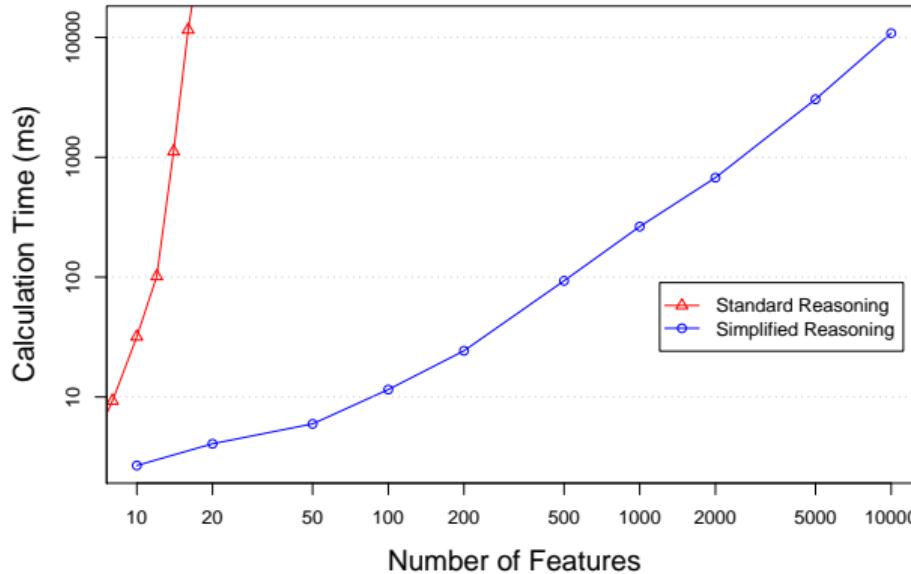
Cross-tree constraints:

- * 10% from number of features
- * Between 2 and 5 variables

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



Scalability - Number of Features



optimizations
are crucial

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

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 Kistner[†], Sebastian Edelborg[‡], 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 decomposing features into smaller components. In order to verify feature modeling, we need to reason about variability and program variants. In this paper, we propose a soundness analysis model, we might want to determine the number of all valid feature combinations or compute specific feature constraints for testing. However, the number of feature combinations opportunities can only exceed about feature combinations, and other opportunities are lost. We propose a technique to identify features into account. Abstract features are features used to describe variability in feature models. They are not part of the implementation level. Using existing feature-model-based reasoning techniques, we can reason about abstract features. Hence, although abstract features represent domain elements that do not affect the generation of a program variant, we can still reason about the existence of abstract features.

Keywords—software product line, program families, feature modeling, feature model, automated analysis.

INTRODUCTION

A software product line is a set of software-intensive systems (program variants) that share common code artifacts [1]. The goal of a software product line is to develop programs simultaneously, by systematically reusing development artifacts. Program variants are distinguished in terms of functional requirements, non-functional requirements, aspects, or characteristics of a software system [2]. Two variants may have several features in common and differ in some others. There are several methods for the line engineering methods, in which program variants can be presented from a common implementation by specifying a common base and adding specific feature techniques, such as conditional compilation [3], program and framework-, 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. Various form of variability models are proposed in literature [21, [8], [9], [10], [11], [12], [13], [14]] and implementations, e.g., a propositional formula in which each feature belongs to a variable and the formula evaluates to true.

We noticed that not all features in typical feature models are used in distinguishable program variants. There are only two types of features in feature models: abstract and concrete. If there does not make any difference in the generated variant code. We denote such features as abstract features. Due to the fact that abstract features do not affect the generation of a program variant, we can still reason about the existence of abstract features.

The set of program variants are not equivalent. Multiple valid feature combinations result in the same generated program.

There are numerous approaches to reason about valid feature combinations in feature models [8]. Nevertheless, there are also many approaches that do not focus on feature models, but on program variants. For example, for combinatorial testing [14], type checking [12]–[15], or verification [16], we can reason about the existence of program variants. There are also approaches that do not focus on feature models, but reason about the relationship of features in program variants. Similarly, for reasoning about non-functional properties of program variants [17], we can reason about abstract features, but can increase measurement effort significantly. Finally, deleting an abstract feature from the feature model does not affect the generated variant code, so it can be considered as feature-model refactoring [18].

Consequently automated analysis of feature models can be used to reason about the existence of abstract features and to reason about program variants. When using automated reasoning methods, the results are at least incomplete or insufficient. As far as we know, there is no work that provides a semantics of feature models as known from literature [19], describing the valid combinations of features and (b) the generation of program variants, i.e., the different variants of a product line. According to our experience, both are needed and complementary to each other. However, in this paper, we focus on the first aspect. In addition to the program-families semantics, we provide a mechanism to translate the program-families semantics in a way that enabling reasoning mechanisms for feature-model variants.

Thüm et al. SPLC11

Even More Remarks on Soundness

Abstract Features in Feature Modeling

Thomas Thüm¹, Christian Kistner¹, Sebastian Edelborg², and Norbert Siegmund¹

¹University of Magdeburg, Germany

²Philipps University Marburg, Germany

Abstract. A software product line is a set of programs variants, typically generated from a common code base. Feature models describe variability in product lines by decomposing features into concrete subfeatures. In feature model reasoning, we need to reason about variability and program variants. In particular, when we want to reason about a feature model, we might want to determine the number of all valid feature models or compute specific feature constraints for testing. However, these reasoning tasks are very challenging because they often require us to reason about many program variants. We argue that, in order to reason about feature models, it is better to reason about abstract features rather than feature combinations, and that this can be made explicit in feature models. We present a technique based on propositional logic that allows us to reason about abstract features rather than feature combinations. In practice, our technique can reduce the number of program variants that have to be considered when reasoning about a feature model. For example, in practice, a program variant might appear multiple times, for example, in product line testing.

Keywords: software product lines, program families, feature modeling, feature model, automated analysis.

1. INTRODUCTION

A software product line is a set of software-intensive systems (program variants) that share common code artifacts [1]. Product lines are often developed and maintained by reuse, for example, by reusing existing code and programs simultaneously, by incrementally refining development artifacts. Program variants are distinguished in terms of features, which are specific requirements, properties, aspects, qualities, or characteristics of a software system [2]. Two variants may have several features in common and differ in others. In feature modeling, we focus on feature line engineering methods, in which program variants can be presented from a common implementation by specifying a feature model. There are many different modeling techniques, such as conditional compilation [3], program and framework, 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,

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

Thüm et al. SPLC11

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

Saving Analysis Effort

Nieke et al. GPCE18

Thüm et al. VariVolution19

Encoding More Than Two Versions

Nieke et al. GPCE18

Young et al. EMSE22

Classifying Edits to Feature Mappings

Bittner et al. FSE22

Encoding Evolution in Feature Model

Seidl et al. SPLC14

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

Validating CNF Translations

Oster et al. SPLC10

Kuiter et al. ASE22

Expressing FM Differences

Acher et al. CAiSE12

Feature Diagram Mutations

Arcaini et al. ICST15

Reuling et al. SPLC15

Validating FM Translations

Feichtinger et al. SPLC22



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



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

EDITED BY:
Proceedings – Volume B

Pascal Arcani, Maurizio H. ter Beek, Gilles Perrouin, Iris Reinhardt-Berger,
Ivan Machado, Silvia Boggia Vergilio, Rick Rabiser, Tao Yue, Xavier Devmey,
Monica Fausto, Hauneari Wachamala



SPLC 2023
Proceedings — Volume A

Sorry Again for Not Reading Your Papers



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

EDITED BY:

Paolo Arcaini, Maurice H. ter Beek, Gilles Perrouin, Iris Reinhardt-Berger,
Ivan Machado, Silvia Iropka Vergilio, Rick Rabiser, Tao Yue, Xavier Devney,
Mélina Faust, Haukestr Wachsmuth



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

Proceedings – Volume A

EDITED BY:

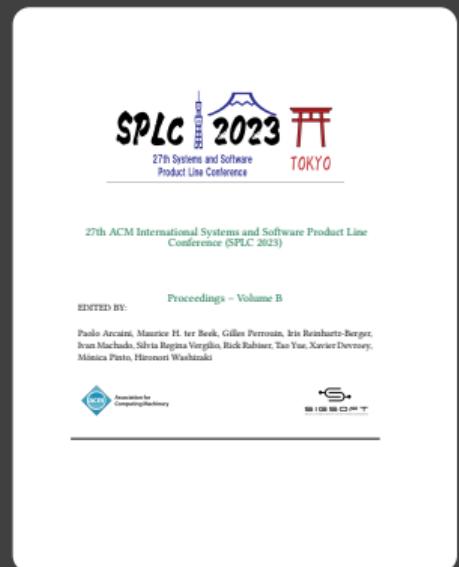
Paolo Arcaini, Maurice H. ter Beek, Gilles Perrouin, Iris Reinhardt-Berger,
Miguel R. Llacer, Christa Schwanninger, Shaukat Ali, Mihai Voiculescu,
Angelo Gargantini, Stefania Gnesi, Maile Lochan, Laura Semini, Hirosoji
Wadzhizaki



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:

We believe this resolves all remaining questions on this topic. No further research is needed.

References

1. ~~all done, no answers, no review done (no) n.n.~~
2. ~~unresolved, unresolved, n.n. (n) n.n.~~
3. ~~n.n., n.n., n.n., n.n. (n) n.n.~~
4. ~~n.n. n.n. n.n. n.n. (n) n.n.~~

JUST ONCE, I WANT TO SEE A RESEARCH PAPER WITH THE GUTS TO END THIS WAY.

Thank You



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

EDITED BY:

Paolo Arcaini, Maurice H. ter Beek, Gilles Perrusin, Iris Reinhardt-Berger,
Ivan Machado, Silvia Iropka Vergilio, Rick Rabiser, Tao Yue, Xavier Devmey,
Monica Fausto, Haunert Wachsmuth



SPLC 2023
Proceedings — Volume A



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

Proceedings — Volume A

EDITED BY:

Paolo Arcaini, Maurice H. ter Beek, Gilles Perrusin, Iris Reinhardt-Berger,
Miguel R. Llacer, Christa Schwaninger, Shaukat Ali, Mihai Vorosan,
Angelo Gargantini, Stefania Gnesi, Maile Lochan, Laura Semini, Hiroshi
Wadzhizaki



SPLC 2023
Proceedings — Volume B

We believe this resolves all remaining questions on this topic. No further research is needed.

References

1. [www, www, www \(w\) w](#)
2. [www, www, www \(w\) w](#)
3. [www, www, www \(w\) w](#)
4. [www, www, www \(w\) w](#)

JUST ONCE, I WANT TO SEE A RESEARCH PAPER WITH THE GUTS TO END THIS WAY.

... for your standing ovations ;o)

Reasoning About Edits to Feature Models (ICSE'09)

1. Introduction

Let's Travel Back to 2007

My Bachelor's Thesis

Motivation

2. Contribution

Edit Categories

Goal and State-of-the-Art

Classification Algorithm

Optimizations

3. Results

Experiment Design

Experiment Results

4. Impact

Conclusion