

Benavides, David (Ed.); Batory, Don (Ed.); Grünbacher, Paul (Ed.)

Research Report

Fourth International Workshop on Variability Modelling of Software-intensive Systems. Proceedings

ICB-Research Report, No. 37

Provided in Cooperation with:

University Duisburg-Essen, Institute for Computer Science and Business Information Systems (ICB)

Suggested Citation: Benavides, David (Ed.); Batory, Don (Ed.); Grünbacher, Paul (Ed.) (2010) : Fourth International Workshop on Variability Modelling of Software-intensive Systems. Proceedings, ICB-Research Report, No. 37, Universität Duisburg-Essen, Institut für Informatik und Wirtschaftsinformatik (ICB), Essen

This Version is available at:

<https://hdl.handle.net/10419/58178>

Standard-Nutzungsbedingungen:

Die Dokumente auf EconStor dürfen zu eigenen wissenschaftlichen Zwecken und zum Privatgebrauch gespeichert und kopiert werden.

Sie dürfen die Dokumente nicht für öffentliche oder kommerzielle Zwecke vervielfältigen, öffentlich ausstellen, öffentlich zugänglich machen, vertreiben oder anderweitig nutzen.

Sofern die Verfasser die Dokumente unter Open-Content-Lizenzen (insbesondere CC-Lizenzen) zur Verfügung gestellt haben sollten, gelten abweichend von diesen Nutzungsbedingungen die in der dort genannten Lizenz gewährten Nutzungsrechte.

Terms of use:

Documents in EconStor may be saved and copied for your personal and scholarly purposes.

You are not to copy documents for public or commercial purposes, to exhibit the documents publicly, to make them publicly available on the internet, or to distribute or otherwise use the documents in public.

If the documents have been made available under an Open Content Licence (especially Creative Commons Licences), you may exercise further usage rights as specified in the indicated licence.



ICB

Institut für Informatik und
Wirtschaftsinformatik

David Benavides
Don Batory
Paul Grünbacher (Eds.)



Fourth International Workshop on Variability Modelling of Software-intensive Systems

ICB-RESEARCH REPORT

Proceedings

Die Forschungsberichte des Instituts für Informatik und Wirtschaftsinformatik dienen der Darstellung vorläufiger Ergebnisse, die i. d. R. noch für spätere Veröffentlichungen überarbeitet werden. Die Autoren sind deshalb für kritische Hinweise dankbar.

The ICB Research Reports comprise preliminary results which will usually be revised for subsequent publications. Critical comments would be appreciated by the authors.

Alle Rechte vorbehalten. Insbesondere die der Übersetzung, des Nachdruckes, des Vortrags, der Entnahme von Abbildungen und Tabellen – auch bei nur auszugsweiser Verwertung.

All rights reserved. No part of this report may be reproduced by any means, or translated.

Authors' Address:

Prof. Dr. Heimo Adelsberger
Dipl.-Wirt.-Inf. Andreas Drechsler

Institut für Informatik und
Wirtschaftsinformatik (ICB)
Universität Duisburg-Essen
Universitätsstr. 9
D-45141 Essen

heimo.adelsberger@icb.uni-due.de
andreas.drechsler@icb.uni-due.de

ICB Research Reports

Edited by:

Prof. Dr. Heimo Adelsberger
Prof. Dr. Peter Chamoni
Prof. Dr. Frank Dorloff
Prof. Dr. Klaus Echtele
Prof. Dr. Stefan Eicker
Prof. Dr. Ulrich Frank
Prof. Dr. Michael Goedicke
Prof. Dr. Tobias Kollmann
Prof. Dr. Bruno Müller-Clostermann
Prof. Dr. Klaus Pohl
Prof. Dr. Erwin P. Rathgeb
Prof. Dr. Albrecht Schmidt
Prof. Dr. Rainer Unland
Prof. Dr. Stephan Zelewski

Contact:

Institut für Informatik und
Wirtschaftsinformatik (ICB)
Universität Duisburg-Essen
Universitätsstr. 9
45141 Essen

Tel.: 0201-183-4041

Fax: 0201-183-4011

Email: icb@uni-duisburg-essen.de

ISSN 1860-2770 (Print)
ISSN 1866-5101 (Online)

Abstract

This ICB Research Report constitutes the Proceedings of the 4th *International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10)*, which was held from January 27–29, 2010 at the Johannes Kepler University Linz.

Table of Contents

1	WELCOME MESSAGE.....	1
2	ORGANIZATION.....	2
3	WORKSHOP FORMAT	5
4	TECHNICAL PROGRAMME	7

1 Welcome Message

Welcome to the 4th *International Workshop on Variability Modelling of Software-intensive Systems: VaMoS'10!* This year we are celebrating *20 years of feature models!*

Previous VaMoS workshops have been held in

- [Limerick \(2007\)](#)
- [Essen \(2008\)](#) and
- [Sevilla \(2009\)](#).

The aim of the VaMoS workshop series is to bring together researchers from various areas of variability modelling in order to discuss advantages, drawbacks and complementarities of the various variability modelling approaches, and to present novel results for variability modelling and management.

Continuing the successful format of the three previous VaMoS workshops, VaMoS 2010 will be a highly interactive event. Each session will be organized in such a way that discussions among the workshop participants will be stimulated. We hope that VaMoS will trigger work on new challenges in variability modelling and thus will help to shape the future of variability modelling research.

VaMoS'10 attracted 50 submissions of authors from 24 countries. Each submission was reviewed by at least three members of the programme committee. Based on the reviews, 16 submissions were accepted as full papers, 6 submissions as short papers and 6 submissions have been accepted as short papers documenting tool demonstrations.

We extend our gratitude to all the people who spent time and energy to make VaMoS a success. VaMoS'10 would not have been possible without their efforts and expertise. We thank Kyo Kang and Krzysztof Czarnecki who accepted our invitation to give keynotes. We cordially thank all the members of the VaMoS programme committee for devoting their time to reviewing the submitted papers, and doing so on time. We are grateful to the people who helped preparing and organizing the event, especially Stephanie Eibensteiner, Karin Gusenbauer, Wolfgang Heider, Roberto E. Lopez-Herrejon, Birgit Kranzl, Kim Lauenroth, and Rick Rabiser. We also thank the student volunteers for their help. Finally, we thank the sponsors of VaMoS: University of Duisburg-Essen, University of Seville, University of Texas at Austin, Johannes Kepler University Linz, the Christian Doppler Laboratory for Automated Software Engineering, the province of Upper Austria, and the City of Linz.

VaMoS is under the distinguished patronage of Dr. Josef Pühringer (Governor of Upper Austria) and Dr. Franz Dobusch (Mayor of Linz).

Enjoy VaMoS 2010 and a beautiful Linz!



David Benavides



Don Batory



Paul Grünbacher

2 Organization

Steering Committee

Ulrich Eisenecker, University of Leipzig, Germany

Patrick Heymans, PReCISE, University of Namur, Belgium

Kyo-Chul Kang, Pohang Univ. of Science and Technology, Korea

Andreas Metzger, University of Duisburg-Essen, Germany

Klaus Pohl, University of Duisburg-Essen, Germany

Honorary Chair

Kyo-Chul Kang, Pohang Univ. of Science and Technology, Korea

General Chair

Paul Grünbacher, Johannes Kepler University Linz, Austria

Program co-chairs

Don Batory, University of Texas, USA

David Benavides, University of Seville, Spain

Organising Committee

Kim Lauenroth, University of Duisburg-Essen, Germany

Roberto E. Lopez-Herrejon, Johannes Kepler University Linz, Austria

Rick Rabiser, Johannes Kepler University Linz, Austria

Program Committee

Vander Alves, University of Brasilia, Brasil
Sven Apel, University of Passau, Germany
Danilo Beuche, Pure::Systems, Germany
Jürgen Börstler, Umeå University, Sweden
Manfred Broy, TU Munich, Germany
Goetz Botterweck, LERO, Ireland
Oscar Díaz, Universidad del País Vasco, Spain
Alessandro Fantechi, Universita' degli Studi di Firenze, Italy
Stefania Gnesi, ISTI-CNR, Italy
Aniruddha Gokhalé, Vanderbilt University, USA
Paul Grünbacher, Johannes Kepler Universitat Linz, Austria
Øystein Haugen, University of Oslo & SINTEF, Norway
Mei Hong, Beijing University, China
Tomoji Kishi, Japan Advanced Institute of Science and Technology
Stan Jarzabek, National University of Singapore
Roberto E. Lopez-Herrejon, Johannes Kepler Universitat Linz, Austria
Tomi Männistö, Helsinki University of Technology, Finland
Vicente Pelechano, Universidad Politécnica de Valencia, Spain
Antonio Ruiz-Cortés, Universidad de Sevilla, Spain
Camille Salinesi, University of Paris 1-Sorbonne, France
Klaus Schmid, University of Hildesheim, Germany
Douglas Schmidt, Vanderbilt University, USA
Steffen Thiel, Furtwangen University of Applied Sciences, Germany
Pim van den Broek, University of Twente, The Netherlands
Frank van der Linden, Philips, The Netherlands
Andrzej Wasowski, IT University of Copenhagen
Matthias Weber, Carmeq GmbH, Germany
Jules White, Vanderbilt University, USA

Additional reviewers

Patrizia Asirelli

Maurice ter Beek

Thorsten Berger

Quentin Boucher

Marcel Böhme

Carlos Cetina

Andreas Classen

Dumitrescu Cosmin

Alessio Ferrari

Alexander Gruler

Alexander Harhurin

Reiko Heckel

Arnaud Hubaux

Martin Johansen

Chang Hwan Peter Kim

Hans Koerber

Neil Loughran

Lami

Alberto Lora

Raul Mazo

Johannes Mueller

Helge Pfeiffer

Fabricia Roos

Marko Rosenmueller

Wolfgang Scholz

Sergio Segura

Steven She

Norbert Siegmund

Bernd Spanfelner

Maria Spichkova

Andreas Svendsen

Pablo Trinidad

Ha Duy Trung

Yinxing Xue

Hongyu Zhang

Wei Zhang

3 Workshop Format

As VaMoS is planned to be a highly interactive event, each session is organized in order to stimulate discussions among the presenters of papers, discussants and the other participants. Typically, after a paper is presented, it is immediately discussed by pre-assigned discussants, after which a free discussion involving all participants follows.

Three particular roles, which imply different tasks, are taken on by the VaMoS attendees:

1) Presenter

A presenter obviously presents his paper but additionally will be asked to take on the role of discussant for the other paper in his session. It is highly desired that – as a presenter – you attend the complete event and take an active part in the discussion of the other papers. Prepare your presentation and bear in mind the available time.

2) Discussant

A discussant prepares the discussion of a paper. Each paper is assigned to two discussants (typically the presenter of the other paper in the same session and a presenter from another session). A discussant's task is to give an unbiased technical review of the paper directly after its presentation. This task is guided by a predefined set of questions that are found in the discussion template provided by the VaMoS organizers.

3) Session Chair

A session chair's tasks are as follows:

Before the session starts:

- Make sure that all presenters and presentations are available.
- Make sure that all discussants are present and that they have downloaded their discussion slides to the provided (laptop) computer.

For each paper presentation:

- Open your session and introduce the presenters.
- Keep track of time and signal the presenters when the end of their time slot is approaching.
- Invite the discussants and organize the individual paper discussions, i.e., ensure that the discussion is structured.
- Close the paper discussion and hand over to the next presenter.

4 Technical Programme

Keynotes

FODA: Twenty Years of Perspective on Feature Modeling <i>Kyo C. Kang</i>	9
---	---

Variability Modeling: State of the Art and Future Directions <i>Krzysztof Czarnecki</i>	11
--	----

Research Papers (Full Papers)

Semistructured Merge in Revision Control Systems <i>Sven Apel, Jörg Liebig, Christian Lengauer, Christian Kästner, William R. Cook</i>	13
---	----

Leveraging Aspect-Connectors to Improve Stability of Product-Line Variabilities <i>Marcelo Dias, Leonardo Tizzei, Ceília Rubira, Alessandro Garcia, Jaejoon Lee</i>	21
--	----

A Formal Semantics for Decision-oriented Variability Modeling with DOPLER <i>Deepak Dhungana, Patrick Heymans, Rick Rabiser</i>	29
--	----

A Deontic Logical Framework for Modelling Product Families <i>Patrizia Asirelli, Maurice H. ter Beek, Stefania Gnesi, Alessandro Fantechi</i>	37
--	----

The Variability Model of The Linux Kernel <i>Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, Krzysztof Czarnecki</i>	45
---	----

A Preliminary Review on the Application of Feature Diagrams in Practice <i>Arnaud Hubaux, Andreas Classen, Marcílio Mendonça, Patrick Heymans</i>	53
--	----

Cardinality-Based Feature Modeling and Model-Driven Engineering: Fitting them Together <i>Abel Gómez, Isidro Ramos</i>	61
---	----

Integrating Automated Product Derivation and Individual User Interface Design <i>Andreas Pleuss, Goetz Botterweck, Deepak Dhungana</i>	69
---	----

Supporting Stepwise, Incremental Product Derivation in Product Line Requirements Engineering <i>Reinhard Stoiber, Martin Glinz</i>	77
---	----

Variability Modelling for Model-Driven Development of Software Product Lines <i>Ina Schaefer</i>	85
---	----

Using Incremental Consistency Management for Conformance Checking in Feature-Oriented Model-Driven Engineering <i>Roberto E. Lopez-Herrejon, Alexander Egyed, Salvador Trujillo, Josune de Sosa, Maider Azanza</i>	93
---	----

The CVM Framework —A Prototype Tool for Compositional Variability Management <i>Andreas Abele, Yiannis Papadopoulos, David Servat, Martin Törngren, Matthias Weber</i>	101
---	-----

Conflict Resolution Strategies During Product Configuration <i>Alexander Nöhner, Alexander Egyed</i>	107
---	-----

Optimizing Non-functional Properties of Software Product Lines by means of Refactorings <i>Norbert Siegmund, Martin Kuhlemann, Mario Pukall, Sven Apel</i>	115
---	-----

Automating the Configuration of Multi Software Product Lines <i>Marko Rosenmüller, Norbert Siegmund</i>	123
--	-----

Variability in Time — Product Line Variability and Evolution Revisited <i>Christoph Elsner, Goetz Botterweck, Daniel Lohmann, Wolfgang Schröder-Preikschat</i>	131
---	-----

Short Papers

Using Collaborations to Encapsulate Features? An Explorative Study <i>Martin Kuhlemann, Norbert Siegmund, Sven Apel</i>	139
Modeling Variability of Augmented Software Product Lines <i>Johannes Müller</i>	143
A Method Based on Association Rules to Construct Product Line Models <i>Alberto Lora-Michiels, Camille Salinesi, Raúl Mazo</i>	147
Measuring the Ability to Form a Product Line from Existing Products <i>Christian Berger, Holger Rendel, Bernhard Rumpe</i>	151
A Custom Approach for Variability Management in Automotive Applications <i>Fabian Kliemann, Georg Rock, Stefan Mann</i>	155
Introducing TVL, a Text-based Feature Modelling <i>Quentin Boucher, Andreas Classen, Paul Faber and Patrick Heymans</i>	159

Tool Demonstrations

XToF – A Tool for Tag-based Product Line Implementation <i>Christophe Gauthier, Andreas Classen, Quentin Boucher, Patrick Heymans, Margaret-Anne Storey, Marcílio Mendonça</i>	163
Tool Support for Evolution of Product Lines through Rapid Feedback from Application Engineering <i>Wolfgang Heider Rick Rabiser</i>	167
Tool Support for Incremental Consistency Checking on Variability Models <i>Michael Vierhauser, Deepak Dhungana, Wolfgang Heider, Rick Rabiser, Alexander Egyed</i>	171
A Support Tool for Domain Analysis <i>Liana Barachisio Lisboa, Vinicius Cardoso Garcia, Silvio Romero de Lemos Meira, Eduardo Santana de Almeida</i>	175
Research Tool to Support Feature Configuration in Software Product Lines <i>Ciarán Cawley, Patrick Healy, Goetz Botterweck, Steffen Thiel</i>	179
SMARTFORM: A Web-based Feature Configuration Tool <i>Wonseok Chae, Timothy L. Hinrichs</i>	183

FODA: Twenty Years of Perspective on Feature Modeling

(Keynote)

Kyo C. Kang

Department of Computer Science and Engineering
Pohang University of Science and Technology (POSTECH)
San 31, Hyoja-Dong, Nam-Gu, Pohang, KOREA, 790-784
E-mail: kck {at} postech.ac.kr
<http://selab.postech.ac.kr/kck/>

Abstract—Feature-oriented domain analysis (FODA) was proposed twenty years ago as a method for systematic discovery and exploitation of commonality across related software systems to support software reuse. Since then, many industrial cases of FODA application have reported, the original model has been extended, and new paradigms such as generative programming and feature-oriented programming have been proposed based on the concept of feature orientation. The research community exploring feature orientation in software development has been growing significantly in recent years as evidenced by the large number of citations of the original work and the researches that followed.

Any software artifacts we create will evolve as new features are added, and existing features are removed or modified, and, therefore, it is important to build softness (which we often call modifiability, adaptability, maintainability, etc.) into software when we design it. FODA stimulates software engineers to think about variability of software they create, and provides a mechanism to codify that knowledge. This variability information is the most critical knowledge for building softness into software, and I believe this is the reason why feature modeling became popular among researchers as well as practitioners. In my talk, I will review the salient features of FODA report and various extensions of the original feature model, and then explore research topics.

Variability Modeling: State of the Art and Future Directions

(Keynote)

Krzysztof Czarnecki

Bank of Nova Scotia / NSERC Industrial Research Chair

University of Waterloo

Department of Electrical and Computer Engineering

200 University Ave. West

Waterloo, ON N2L 3G1, Canada

e-mail: czarnecki@acm.org

www.gsd.uwaterloo.ca/kczarneck

www.generative-programming.org

Abstract—Feature modeling started 20 years ago as a simple, yet very appealing and intuitive form of variability modeling. Over time researchers proposed numerous extensions to accommodate the needs of many real-world applications. As a result, a large number of distinct forms of feature modeling exist in the literature. I will present an ongoing effort to design a unified feature modeling language that supports both Boolean and rich cardinality-based feature models with natural simplicity. The effort is inspired by real-world variability modeling languages and is driven by a diverse set of applications. Among others, I will attempt to answer the following question: What is the remaining essence of feature modeling, once a feature modeling language includes most of the mechanisms present in other structural modeling languages, such as class modeling? I will also discuss progress in automated tools and techniques for performing operations on feature models, such as configuration, analysis, and synthesis. I will close with a discussion of challenges and future directions in variability modeling, including the challenge of mapping features to and representing them in other software artifacts.

Any software artifacts we create will evolve as new features are added, and existing features are removed or modified, and, therefore, it is important to build softness (which we often call modifiability, adaptability, maintainability, etc.) into software when we design it. FODA stimulates software engineers to think about variability of software they create, and provides a mechanism to codify that knowledge. This variability information is the most critical knowledge for building softness into software, and I believe this is the reason why feature modeling became popular among researchers as well as practitioners. In my talk, I will review the salient features of FODA report and various extensions of the original feature model, and then explore research topics.

Semistructured Merge in Revision Control Systems

Sven Apel, Jörg Liebig, Christian Lengauer
 Dept. of Informatics and Mathematics
 University of Passau
 {apel,joliebig,lengauer}@fim.uni-passau.de

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

William R. Cook
 Dept. of Computer Sciences
 University of Texas at Austin
 wcook@cs.utexas.edu

Abstract—Revision control systems are a major means to manage versions and variants of today’s software systems. An ongoing problem in these systems is how to resolve conflicts when merging independently developed revisions. Unstructured revision control systems are purely text-based and solve conflicts based on textual similarity. Structured revision control systems are tailored to specific languages and use language-specific knowledge for conflict resolution. We propose semistructured revision control systems to inherit the strengths of both classes of systems: generality and expressiveness. The idea is to provide structural information of the underlying software artifacts in the form of annotated grammars, which is motivated by recent work on software product lines. This way, a wide variety of languages can be supported and the information provided can assist the resolution of conflicts. We have implemented a preliminary tool and report on our experience with merging Java artifacts. We believe that drawing a connection between revision control systems and product lines has benefits for both fields.

I. INTRODUCTION

Revision control systems (a.k.a. version control systems) have a long tradition in software engineering [1], [2]. On the one hand, they are used in virtually every substantial software project in industry. On the other hand, they have also attracted much attention in academia. Revision control systems are a major means to manage versions and variants of today’s software systems. A programmer creates a revision of a software system by deriving it from the base system or from another revision; a revision can be developed and evolve in isolation; and it can be merged again with the base system or another revision. A major problem of revision control is how to resolve merge conflicts that are caused by concurrent changes.

In the recent years, two classes of revision control systems have emerged: (1) revision control systems that operate on plain text and (2) revision control systems that operate on more abstract and structured document representations. The first class is used widely in practice, since such systems are typically language-independent (i.e., they work with every software artifact that can be represented with text). Some widely used systems of this class are CVS¹, Subversion², Git³, and Mercurial⁴. Henceforth, we call them *unstructured revision control systems*. A problem is that, when conflicts occur, the unstructured revision control system has no knowledge

of the structure of the underlying software artifacts, which makes it difficult to resolve certain kinds of conflicts, as we will illustrate.

The second class is explored mainly in academia with the goal of solving the problems of unstructured revision control systems with the conflict resolution. The idea is to use the structure and semantics of the software artifacts being processed to resolve merge conflicts automatically [3]. These systems operate on abstract syntax trees or similar representations instead of on plain program text. A drawback is that, aiming at a particular language’s syntax or semantics, they sacrifice language independence. Henceforth, we call these systems *structured revision control systems*.

Apparently, there is a trade-off between generality and expressiveness of revision control systems. A revision control system is general, if it works with many different kinds of software artifacts. It is expressive if it is able to handle as many merge conflicts as possible automatically. Inspired by the trade-off between generality and expressiveness, we propose a new class of revision control systems, called *semistructured revision control systems*, that inherits the strengths but not the weaknesses of structured and unstructured revision control systems. The idea is to increase the amount of information a revision control system has at its disposal to resolve conflicts, while maintaining generality in the sense that many languages are supported. In particular, we concentrate on the merge process, so we speak of *semistructured merge*.

Our proposal is based on previous work on language-independent feature composition in software product line engineering [4], [5]. We noticed a strong similarity between software composition tools and software merging techniques used in revision control systems, which we exploit in our proposal. In a nutshell, we extend an existing feature composition tool infrastructure, called FEATUREHOUSE, to enable it to merge different revisions of a software system based on the structure of the software artifacts involved. Users can plug new languages into FEATUREHOUSE by providing a formal specification of their languages’ syntax (i.e., the grammar) enriched with semantic information. While this approach is not entirely language-independent, it is still quite general in that new languages can be integrated easily by providing their grammars. If, for whatever reason, there is no grammar available for a certain language, a programmer can parse corresponding software artifacts line by line, which would be effectively the unstructured approach.

¹<http://www.cvshome.org/eng/>

²<http://subversion.tigris.org/>

³<http://git-scm.com/>

⁴<http://mercurial.selenic.com/>

```

1 import java.util.LinkedList;
2 public class Stack<T> {
3     private LinkedList<T> items = new LinkedList<T>();
4     public void push(T item) {
5         items.addFirst(item);
6     }
7     public T pop() {
8         if(items.size() > 0) return items.removeFirst();
9         else return null;
10    }
11 }

```

Fig. 1. A simple stack implementation in Java.

Learning from product line engineering pays off in the development of revision control systems, as we will demonstrate. But also the reverse is interesting. Revision control systems are used widely in practice to manage versions and variants. We and others believe that drawing a connection between revision control systems and software product lines also provides insights for software product line engineers, especially with regard to real-world application scenarios [6].

In the remainder, we analyze the trade-off between generality and expressiveness of structured and unstructured merge. Based on the analysis, we derive our proposal of semistructured merge. Furthermore, we offer a preliminary tool that demonstrates the principal applicability of the approach, and we report on first experiences with it.

II. CONFLICTS IN REVISION CONTROL – BACKGROUND AND RELATED WORK

There is a large body of work on revision control systems [1], [2] and conflict resolution in software merging [3]. We concentrate on aspects relevant for our proposal. The purpose of a revision control system is to manage different revisions of a software system. Usually, revisions are derived from a base program or from other revisions. By branching the development line, a programmer can create independent revisions, which can be changed and evolve in isolation (e.g., to add and test new features). Finally, independent revisions can be merged again with the base program or with other revisions, which may have been changed in the meantime.

The key issue we address in our work is merge conflict resolution. When two revisions have evolved independently, conflicts may occur while merging them. A major goal of research in this area is to empower revision control systems to resolve merge conflicts automatically [3]. First, we illustrate the problem of conflict resolution in unstructured merge. Then, we highlight some mechanisms of structured merge that enable them to resolve conflicts better than unstructured merge.

A. Unstructured Merge

To illustrate the conflict resolution problem, we use the running example of a simple stack implementation, as shown in Figure 1. Henceforth, we call this program the base program or simply STACK. It contains a class `Stack` that contains a field `items` and the two methods `push` and `pop`.

```

1 import java.util.LinkedList;
2 public class Stack<T> {
3     private LinkedList<T> items = new LinkedList<T>();
4     public void push(T item) {
5         items.addFirst(item);
6     }
7     public T top() {
8         return items.getFirst();
9     }
10    public T pop() {
11        if(items.size() > 0) return items.removeFirst();
12        else return null;
13    }
14 }

```

Fig. 2. A revision of the stack implementation that adds method `top`.

```

1 import java.util.LinkedList;
2 public class Stack<T> {
3     private LinkedList<T> items = new LinkedList<T>();
4     public void push(T item) {
5         items.addFirst(item);
6     }
7     public int size() {
8         return items.size();
9     }
10    public T pop() {
11        if(items.size() > 0) return items.removeFirst();
12        else return null;
13    }
14 }

```

Fig. 3. A revision of the stack implementation that adds method `size`.

Now, suppose a programmer would like to add a new feature `TOP`, but would like to develop the feature in its own branch, independently of the main branch (i.e., base program). To this end, the programmer creates a branch with a new revision `TOP`. Furthermore, suppose another programmer adds subsequently a feature `SIZE` directly to the main branch⁵ by creating a corresponding revision of the base program. Figure 2 and Figure 3 present code for the two revisions, each of which add a new method to class `Stack`. Finally, suppose that, at some point in time, the two branches are merged again to combine both revisions including the new features.

Merging the two branches involves merging the two revisions `TOP` and `SIZE` on the basis of the common ancestor, the base program `STACK`. This process is also called a *three-way merge* because it involves three programs or documents [2]. In our example, the merge process reports a conflict that cannot be resolved automatically with unstructured merge. Figure 4 illustrates the output of the Linux merge tool for this example. The figure shows that the merge process is not able to merge the two new methods `top` and `size` such that both can be present in the merged program.

This example is very simple but it illustrates already the problems of unstructured merge. An unstructured merge tool operates solely on the basis of text lines or tokens. It identifies new text fragments with regard to the common ancestor (base program) and stores the common fragments before and after

⁵Note that the programmer could develop feature `SIZE` in any other branch but, for simplicity, we assume that the main branch is used.

merge_unstruct(TOP, STACK, SIZE)

```

1 import java.util.LinkedList;
2 public class Stack<T> {
3     private LinkedList<T> items = new LinkedList<T>();
4     public void push(T item) {
5         items.addFirst(item);
6     }
7     <<<<<< Top/Stack.java
8     public T top() {
9         return items.getFirst();
10    }
11    =====
12    public int size() {
13        return items.size();
14    }
15    >>>>>> Size/Stack.java
16    public T pop() {
17        if (items.size() > 0) return items.removeFirst();
18        else return null;
19    }
20 }

```

Fig. 4. Output of the Linux merge tool when merging revision TOP and SIZE with the base program.

merge_struct(TOP, STACK, SIZE)

```

1 import java.util.LinkedList;
2 public class Stack<T> {
3     private LinkedList<T> items = new LinkedList<T>();
4     public void push(T item) {
5         items.addFirst(item);
6     }
7     public T top() {
8         return items.getFirst();
9     }
10    public int size() {
11        return items.size();
12    }
13    public T pop() {
14        if (items.size() > 0) return items.removeFirst();
15        else return null;
16    }
17 }

```

Fig. 5. Merging TOP and SIZE without conflicts.

the new fragments. If the two revisions change or extend text in the same region, the system reports a conflict, i.e., it is not able to decide how to merge the changes or extensions. In our example, the merge tool knows that two independent text fragments (which actually implement the two methods `top` and `size`) are added to the same location of the base program (which is enclosed by the two fragments that implement the methods `push` and `pop`). The problem is that the unstructured merge tool does not know that these fragments are methods and that a merge of the two is actually straightforward, as we illustrate next.

Why is an unstructured revision control system not able to resolve the conflict that occurs when merging the revisions TOP and SIZE? As indicated before, an unstructured merge tool does not know that the two fragments implement Java methods whose order does not matter within a class declaration. If the tool knew that the base program and the two revisions are actually Java programs, then it would be able to solve the conflict automatically. There are actually two ways to resolve the conflict: include method `top` first and then method `size` (shown in Figure 5), or vice versa.

Revision SERIALIZABLE

```

1 import java.util.LinkedList;
2 import java.io.Serializable;
3 public class Stack<T> implements Serializable {
4     private static final long serialVersionUID = 42;
5     ...
6 }

```

Fig. 6. Revision that makes Stack objects serializable.

Revision FLUSHABLE

```

1 import java.util.LinkedList;
2 import java.io.Flushable;
3 public class Stack<T> implements Flushable {
4     ...
5     public void flush() { ... }
6 }

```

Fig. 7. Revision that makes Stack objects flushable.

B. Structured Merge

Figure 5 illustrates a very simple example of taking advantage of information on the syntax and semantics of the programs and revisions involved in the merge process. In the past, many tools have been proposed that leverage this kind of information to resolve as many conflicts as possible [3]. Westfechtel and Buffenbarger pioneered this field by proposing tools that incorporate structural information such as the context-free and context-sensitive syntax in the merge process [7], [8]. Researchers proposed a wide variety of structural comparison and merge tools including tools specific to Java [9] and C++ [10]. Some tools even consult additionally semantic information [11]–[13].

Let us illustrate the abilities of structured merge by a further example. Suppose we have the base stack implementation and we create two independent revisions, one to develop feature SERIALIZABLE that enables stack objects to be serialized and another to develop feature FLUSHABLE that allows programmers to flush the elements of the stack to a data stream. Figure 6 and Figure 7 depict excerpts of the two revisions.

Merging the two revisions with the base program using unstructured merge causes two conflicts. First, the system is not able to merge the two new import statements and, second, it is not able to merge the two implements clauses of the two revisions. Figure 8 shows the conflicts as reported by the Linux merge tool.

A structured revision control system that knows that the base program and the revisions are written in Java is able to resolve the conflicts automatically and produces the desired result (i.e., the imports are placed one after the other and the implements clauses are concatenated), as shown in Figure 9.

Beside the conflicts we have seen so far, there are many more conflicts that can be resolved by structured revision control systems on the basis of language-specific knowledge. For example, a `for` loop in Java consists of a header and a body, and the header consists of three parts. This information is useful when two revisions modify disjoint parts of the header.

```

merge_unstruct(SERIALIZABLE, STACK, FLUSHABLE)
1 import java.util.LinkedList;
2 <<<<<<< Serializable/Stack.java
3 import java.io.Serializable;
4 =====
5 import java.io.Flushable;
6 >>>>>> Flushable/Stack.java
7 <<<<<<< Serializable/Stack.java
8 public class Stack<T> implements Serializable {
9     private static final long serialVersionUID = 42;
10 =====
11 public class Stack<T> implements Flushable {
12 >>>>>> Flushable/Stack.java
13     private LinkedList<T> items = new LinkedList<T>();
14     public void push(T item) {
15         items.addFirst(item);
16     }
17     public T pop() {
18         if (items.size() > 0) return items.removeFirst();
19         else return null;
20     }
21     public void flush() { ... }
22 }

```

Fig. 8. Output of the Linux merge tool merging revision SERIALizable and FLUSHABLE with the base program.

```

merge_struct(SERIALIZABLE, STACK, FLUSHABLE)
1 import java.util.LinkedList;
2 import java.io.Serializable;
3 import java.io.Flushable;
4 public class Stack<T> implements Serializable, Flushable {
5     private static final long serialVersionUID = 42;
6     private LinkedList<T> items = new LinkedList<T>();
7     public void push(T item) {
8         items.addFirst(item);
9     }
10     public T pop() {
11         if (items.size() > 0) return items.removeFirst();
12         else return null;
13     }
14     public void flush() { ... }
15 }

```

Fig. 9. The desired result of merging revision SERIALizable and FLUSHABLE with the base program.

C. Generality vs. Expressiveness

The previous discussion reveals that there is a trade-off between generality and expressiveness of revision control systems. Unstructured revision control systems are very general. They can be used with every kind of (textual) software artifact. However, they are not able to resolve conflicts that require knowledge on the language of the artifacts involved. Typically, a structured revision control system is tailored to a particular language. So, it would be possible to build a revision control system for Java that can resolve the conflicts we have discussed so far and, in addition, many other conflicts. However, such a system would be useless in a setting in which a software system consists of artifacts written in many different languages, most notably software product lines [4], [14].

The trade-off motivates us to explore the space between structured and unstructured revision control systems. Can we invent a system that is able to handle a wide variety of software artifacts and that has enough information on these artifacts to resolve a reasonable number of conflicts automatically? A trivial solution would be to develop a structured revision

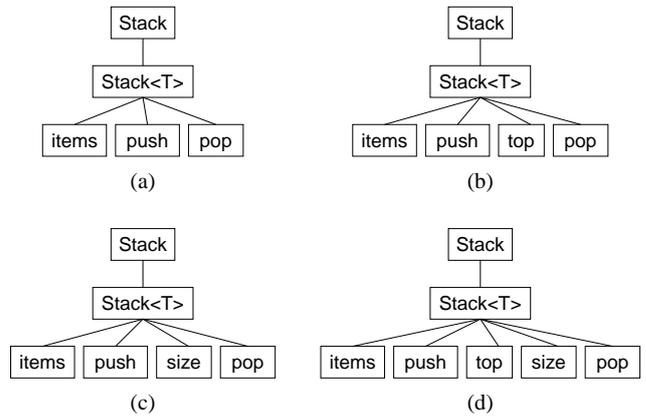


Fig. 10. Different revisions of the stack example represented as program structure trees.

control system for every artifact type that occurs in a software project. A problem with this naive approach is that it is very tedious and error-prone. Moreover, in many cases, not all artifact types can be anticipated; in times where people invent their own domain-specific languages and document formats, this approach is simply infeasible.

III. SEMISTRUCTURED MERGE

The basic idea of semistructured revision control systems—which is much like in structured revision control systems—is to represent software artifacts as trees and to provide information on how the nodes of a certain type (e.g., methods or classes) and their subtrees are merged. We call such a tree, which is essentially a parse tree, a *program structure tree* (a.k.a. *feature structure tree*) [5]. In Figure 10(a), we show a simplified program structure tree of the base program STACK, and, in Figure 10(b) and Figure 10(c), we show simplified program structure trees of TOP and SIZE. It is important to note that not all structural information is represented in the tree. For example, there are no nodes that represent statements or expressions. But the structural information is not lost; it is contained as plain text in the leaves (not shown). So a program structure tree is not necessarily a full parse tree but abstracts from some details and represents them as plain text.

The choice of which kind of structural element is represented by a distinct node depends on the expressiveness which we want to attain with semistructured merge. Let us explain this choice by means of the stack example. Taking the three program structure trees as input, a merge tool can produce the desired output just by superimposing the trees, as shown in Figure 10(d). Why does this algorithm work?⁶ It works because the order of methods does not matter. If the two revisions added methods with identical signatures, the tool would have to merge the statements of their bodies. This would be more difficult since their order matters (and statements

⁶Note that the structured merge tools of Westfechtel [7] and Buffenbarger [8] are not able to resolve this kind of conflict (personal communication with Westfechtel). However, in principle they would have enough information to do so.

do not have unique names). Even with all knowledge on the Java language, there are always cases in which we cannot say how to merge sequences of statements. This is the reason why we choose to represent methods as leaves and their statements as sole text content; in other languages, we may choose differently.

In the example of Figure 8, we see that unstructured merge is not able to combine the differing implements clauses of two revisions of a class. With semistructured (and structured) merge, we are able to achieve this because we know that lists of types can be concatenated.⁷ But what do we do with program elements of which we do not know how to merge them, such as method bodies with statements? The answer is simple: We represent the elements as plain text and use conventional unstructured merge. That is, if a conflict occurs inside a method body, we cannot resolve it automatically—much like in unstructured merge.

So, semistructured merge is more expressive than unstructured merge because certain conflicts can be resolved automatically; but it is less expressive than truly structured merge because some content is treated as plain text. The question that arises is: Why not use structured merge altogether? The answer is: This way we lose more and more generality, as we explain next.

A. Balancing Generality and Expressiveness

The ability of semistructured merge to resolve the conflicts which we have discussed so far is based on the observation that the order of certain elements, e.g., of classes, interfaces, methods, imports, implements and throws list elements, and so on, does not matter. We call those conflicts *ordering conflicts*. A merge algorithm that just resolves ordering conflicts automatically is simpler to define than a full structural merge. A semistructured merge uses an *abstraction* of the structure of the document, where the abstraction has just enough information to identify ordered items and resolve conflicts.

Thus, our system consists of two parts: (1) a generic engine that knows how to identify and resolve ordering conflicts and (2) a small abstract specification—for each artifact type—of the program’s or document’s elements of which the order does not matter. The abstract specification of a document structure is given by an annotated grammar of the language. Most of the difficult work is done by the generic merge engine, using the grammar as a guide. This architecture makes it relatively easy to include new languages by providing proper abstract specifications. For example, the order of data type declarations in a Haskell program or of functions in a Python program does not matter.

To illustrate the role of annotations, consider the excerpt of a simplified Java grammar in Figure 11. It contains a set of production rules. For example, the rule `ClassDecl` defines the structure of classes containing fields (`FieldDecl`), constructors (`ClassConstr`), and methods (`MethodDecl`).

⁷Again, the structured merge tools of Westfechtel [7] and Buffenbarger [8] are not able to resolve this kind of conflict, but this should be possible in principle.

```

1 @FSTNonTerminal(name="{Type}")
2 ClassDecl : "class" Type ImplList "{"
3   (FieldDecl)* (ClassConstr)* (MethodDecl)*
4   "}";
5 ...
6 @FSTNonTerminal(name="ImplClause")
7 ImplList : "implements" @LIST Type ("," @LIST Type)*
8 @FSTTerminal(name="{<ID>} ({ParamList})")
9 MethodDeclaration :
10   Type <ID> "(" (ParamList)? ")" "{"
11   (Statement)*;
12   "}";
13 @FSTTerminal(name="{TOSTRING}")
14 Type : ...

```

Fig. 11. An excerpt of a simplified Java grammar with semantic annotations.

Production rules may be annotated with `@FSTNonTerminal` and `@FSTTerminal`. The former annotation defines that (1) elements corresponding to the rule are represented as nodes in corresponding program structure tree, (2) there may be subnodes, and (3) the order of elements or nodes is arbitrary. In our example, we annotate the rule for class declarations with `@FSTNonTerminal`⁸ because classes may contain further classes, methods, and so on, and the order of classes in a file or package may vary. The `@FSTTerminal` annotation is like the `@FSTNonTerminal` annotation except that subelements are represented as plain text. We annotate the rule for method declarations in this way because the order of methods may vary but their inner statements are represented by plain text, as explained before. A further interesting example is the rule for implements lists. This rule is annotated with `@FSTNonTerminal`, so the order of elements (i.e., type names) of an implements list may vary. However, for a parser, it is difficult to recognize that the elements form really a list, which is basically due to the grammar’s treatment of the commas between the elements. The inner annotation `@LIST` passes exactly this information to the generated parser.

Beyond ordering conflicts we can imagine many ways to use annotations for conflict resolution. For example, we could use annotations to specify how the parts of a `for` loop header are merged. In this case, the order of the parts matters, so the annotations would have to be of a different kind. In further work, we will explore the potential of our approach to resolve other kinds of conflicts.

As we have illustrated, an annotated grammar contains sufficient information to guide a language-independent revision control system in merging Java artifacts. But how does this approach facilitate generality? Indeed, for a language to be supported, we need some information in the form of an annotated grammar, so the tool is not entirely language-independent. But such a grammar is easily provided, since standard grammars in Backus-Naur-Form are available on the Web for many languages, and adding annotations is a matter of hours, at most. Actually, we do not even need the entire grammar of a language, but only the part that is concerned with elements whose order is flexible. We have

⁸The annotation parameter `name` is used to assign a name to the corresponding nodes in the program structure tree.

been quite successful using such a mechanism in feature composition in software product line engineering [4] and we expect to reproduce the success of applying such a mechanism in revision control systems.

To summarize, semistructured merge is more expressive than unstructured merge, since certain conflicts can be resolved automatically based on information on the underlying languages; and semistructured merge is more general than structured merge, since a wide variety of languages can be supported solely on the basis of providing an annotated grammar, which needs to be done only once per language. If, for whatever reason, there is no information available on a given language, semistructured merge behaves exactly like unstructured merge, parsing the corresponding software artifact line by line.

IV. IMPLEMENTATION AND EXPERIENCE

We have implemented a first prototype of a semistructured merge tool, called FSTMERGE, which is able to resolve ordering conflicts.⁹ FSTMERGE takes advantage of our existing tool infrastructure FEATUREHOUSE, as illustrated in Figure 12. The tool FSTGENERATOR generates almost all code that is necessary for the integration of a new language into FSTMERGE. FSTGENERATOR expects the grammar of the language in a proprietary format, called FEATUREBNF, of which we have shown already an example in Figure 11. Using a grammar written in FEATUREBNF, FSTGENERATOR generates an LL(k) parser (which produces program structure trees) and a corresponding pretty printer, which are then integrated into FSTMERGE. After the generation step, FSTMERGE proceeds as follows: (1) the generated parser receives the base program and two revisions written in the target language and produces for each program a program structure tree; (2) FSTMERGE performs the semistructured merge as explained before (the trees are superimposed and a conventional unstructured merge is applied to the leaves); (3) the generated pretty printer writes the merged revisions to disk.

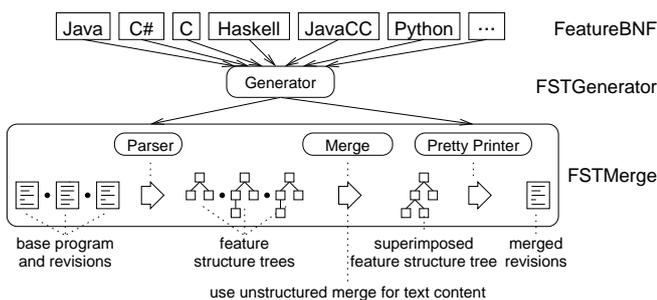


Fig. 12. The architecture of FEATUREHOUSE.

So far, we have used FSTMERGE only with Java programs and we concentrated on ordering conflicts (including merging classes containing methods, fields, implements lists). But,

⁹FSTMERGE and some examples can be downloaded with the FEATUREHOUSE distribution: <http://www.fosd.de/fh/>

due to our experience with FEATUREHOUSE in software product line engineering [4], [5], [15]–[18], we expect that integrating further languages is very easy. In fact, we have already developed the annotated grammars of several further languages including C, C#, JavaCC, and Haskell—what is missing are case studies. The more interesting issue is whether semistructured merge can play to its strengths in real software projects. It is clear that semistructured merge is able to resolve more conflicts than unstructured merge. But how frequent are such conflicts? For example, how often are methods added independently to the same region? How often are implements lists changed? Currently, we cannot provide answers. Although there is some evidence that revisions often involve additions of larger structures such as entire functions [19], we need a substantial set of data to answer the question definitely. From the theoretical point of view, semistructured merge is very interesting, not least because, by means of playing with annotations, we can adjust the way the merge tool works. However, the impact on practical revision control remains to be evaluated. A first step is to analyze the kind and frequencies of conflicts in different software projects incorporating different kinds of software artifacts.

V. CONCLUSION AND OPEN ISSUES

Both unstructured and structured revision control systems have strengths and weaknesses. The former are very general but cannot resolve certain kinds of conflicts. The latter are typically tailored to specific languages and can thus resolve conflicts better than the former. To profit from both worlds, we have proposed semistructured merge, which is inspired by our previous work on software product lines. Developers provide information on the artifact languages in the form of annotated grammars. This way, a wide variety of different languages can be supported while taking advantage of the provided information during the merge process. We have implemented a preliminary tool and plugged in support for the Java language. Whereas integrating further languages is straightforward, it is interesting to explore whether semistructured merge can play to its strengths in practical software engineering. Finally, it is interesting to study the commonalities and differences of software product lines and revision control systems. We have taken a first step and we believe that, in the future, both fields will converge.

We see three interesting open issues of our approach. The first issue is that semistructured merge (much like structured merge) relies on structural information, so the revisions must be syntactically correct. Whereas it is best practice to commit only correct programs or documents, this is not a strict requirement of today's (unstructured) revision control systems. In such cases, the artifacts involved have to be parsed as plain text such that semistructured merge behaves exactly like unstructured merge. It is interesting to explore whether in such cases syntactically correct fragments can be represented by program structure trees and only the incorrect fragments as plain text.

A further issue is the role of refactorings. So far we have not addressed changes like the renaming of methods or classes. For example, in semistructured merge, a rename method refactoring would result in two different methods (the original method and the renamed method), without reporting a conflict. It is debatable if this is the desired behavior. On the other hand, we believe that structural information of whatever kind is of a great value in the presence of refactoring. One can even imagine to tune the kind of information that is passed to the merge tool, e.g., information on references between program elements instead of ordering information. We will explore this issue in further work.

Finally, it would be interesting to explore how type information can be used to resolve conflicts. The problem is that type systems are typically tailored to specific languages and thus would undermine generality. However, researchers begin to think about cross-language and language-independent type systems [20]–[22]. In the future, it may be possible to use such a type system for conflict resolution in semistructured revision control systems.

ACKNOWLEDGMENTS

We thank Don Batory for fruitful discussions on the potential of semistructured merge. This work has been supported in part by the German Research Foundation (DFG), project number AP 206/2-1.

REFERENCES

- [1] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys (CSUR)*, vol. 30, no. 2, pp. 232–282, 1998.
- [2] B. O'Sullivan, "Making Sense of Revision-Control Systems," *Communications of the ACM (CACM)*, vol. 52, no. 9, pp. 56–62, 2009.
- [3] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 5, pp. 449–462, 2002.
- [4] S. Apel, C. Kästner, and C. Lengauer, "FeatureHouse: Language-Independent, Automated Software Composition," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2009, pp. 221–231.
- [5] S. Apel and C. Lengauer, "Superimposition: A Language-Independent Approach to Software Composition," in *Proceedings of the International Symposium on Software Composition (SC)*, ser. Lecture Notes in Computer Science, vol. 4954. Springer-Verlag, 2008, pp. 20–35.
- [6] M. Staples and D. Hill, "Experiences Adopting Software Product Line Development without a Product Line Architecture," in *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*. IEEE Computer Society, 2004, pp. 176–183.
- [7] B. Westfechtel, "Structure-Oriented Merging of Revisions of Software Documents," in *Proceedings of the International Workshop on Software Configuration Management (SCM)*. ACM Press, 1991, pp. 68–79.
- [8] J. Buffenbarger, "Syntactic Software Merging," in *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops on Software Configuration Management*, ser. Lecture Notes in Computer Science, vol. 1005. Springer-Verlag, 1995, pp. 153–172.
- [9] T. Apiwattanapong, A. Orso, and M. Harrold, "JDiff: A Differencing Technique and Tool for Object-Oriented Programs," *Automated Software Engineering*, vol. 14, no. 1, pp. 3–36, 2007.
- [10] J. Grass, "Cdiff: A Syntax Directed Differencer for C++ Programs," in *Proceedings of the USENIX C++ Conference*. USENIX Association, 1992, pp. 181–193.
- [11] V. Berzins, "Software Merge: Semantics of Combining Changes to Programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 6, pp. 1875–1903, 1994.
- [12] D. Jackson and D. Ladd, "Semantic Diff: A Tool for Summarizing the Effects of Modifications," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 1994, pp. 243–252.
- [13] D. Binkley, S. Horwitz, and T. Reps, "Program Integration for Languages with Procedure Calls," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 4, no. 1, pp. 3–35, 1995.
- [14] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE Transactions on Software Engineering (TSE)*, vol. 30, no. 6, pp. 355–371, 2004.
- [15] S. Apel, C. Lengauer, B. Möller, and C. Kästner, "An Algebra for Features and Feature Composition," in *Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST)*, ser. Lecture Notes in Computer Science, vol. 5140. Springer-Verlag, 2008, pp. 36–50.
- [16] S. Apel, F. Janda, S. Trujillo, and C. Kästner, "Model Superimposition in Software Product Lines," in *Proceedings of the International Conference on Model Transformation (ICMT)*, ser. Lecture Notes in Computer Science, vol. 5563. Springer-Verlag, 2009, pp. 4–19.
- [17] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer, "Feature (De)composition in Functional Programming," in *Proceedings of the International Conference on Software Composition (SC)*, ser. Lecture Notes in Computer Science, vol. 5634. Springer-Verlag, 2009, pp. 9–26.
- [18] S. Apel, C. Lengauer, D. Batory, B. Möller, and C. Kästner, "An Algebra for Feature-Oriented Software Development," Department of Informatics and Mathematics, University of Passau, Tech. Rep. MIP-0706, 2007.
- [19] G. Stoyile, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "Mutatis Mutandis: Safe and Predictable Dynamic Software Updating," in *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2005, pp. 183–194.
- [20] M. Grechanik, D. Batory, and D. Perry, "Design of Large-Scale Polylingual Systems," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2004, pp. 357–366.
- [21] S. Apel and D. Hutchins, "An Overview of the gDeep Calculus," Department of Informatics and Mathematics, University of Passau, Tech. Rep. MIP-0712, 2007.
- [22] S. Apel and D. Hutchins, "A Calculus for Uniform Feature Composition," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2010.

Leveraging Aspect-Connectors to Improve Stability of Product-Line Variabilities

Marcelo Dias ^{#1}, Leonardo Tizzei ^{#2}, Cecília Rubira ^{#3}, Alessandro Garcia ^{*4}, Jaejoon Lee ^{†5}

[#]*Institute of Computing, University of Campinas
Campinas, Brazil*

¹marcelo.dias@students.ic.unicamp.br

²tizzei@ic.unicamp.br

³cmrubira@ic.unicamp.br

^{*}*Informatics Department, PUC-Rio
Rio de Janeiro, Brazil*

⁴alessandro.garcia@puc-rj.br

[†]*Computing Department, Lancaster University
Lancaster, UK*

⁵j.lee@comp.lancs.ac.uk

Abstract—One of the design goals of Product Line Architectures (PLAs) is to remain stable while accommodating changes of stakeholder’s requirements. However, the stability of a PLA is largely dependent on how modularized are the decisions crosscutting multiple architectural variation points. Their scattered implementation often leads to a number of side effects, such as architecture-wide modifications. This paper proposes a novel component model to encapsulate architectural variation points inside aspect-connectors, called *Connector-VPs*. Our component model addresses limitations of emerging aspect-oriented models, such as XPIs, which do not allow a modular implementation of crosscutting variability decisions in a PLA. The role of a *Connector-VP* is both binding aspectual-level components to base-level ones, and isolating crosscutting decisions at architectural variation points. We have evaluated the PLA stability designed using our solution in the presence of heterogeneous evolutionary scenarios in the context of component-based PLAs. The results show that our solution tends to promote superior PLA resilience on these scenarios.

Index Terms—Component-based software development; Aspect-Oriented Programming; Software Product Lines; Software Architecture; Architectural Variability;

I. INTRODUCTION

Software product line (SPL) engineering aims at improving development efficiency for families of software systems in a given domain [7]. This concept promotes large-scale reuse through a Product Line Architecture (PLA) that is common to a variety of similar products in terms of their architectural elements. The combination of SPL and Component-based Development (CBD) is a well-known technique to rapidly and efficiently derive products from a set of reusable assets [3]. In the CBD, software systems are developed composing interoperable and reusable blocks called *software components* [24]. A component-based PLA fosters explicit representation of component specification and contributes to reduce coupling and increase cohesion, thereby improving SPL modularity and evolvability [4].

In the context of component-based PLAs, an architectural

variation point is a place on the PLA where decisions are made to derive different products [25]. Architectural variation points are associated with non-kernel features (optional or alternative ones). Evolution scenarios involving non-kernel feature cause changes in architectural variation points. These scenarios usually involve inclusion of non-kernel features and/or changing kernel features into non-kernel ones. In this context, it is important for organizations to achieve design stable PLAs and achieving a controlled evolution of architectural variation points lies at the heart of it. A *stable* PLA means that it can endure evolutionary changes by sustaining its modularity properties.

One of the modern approaches to support enhanced modularity is Aspect-Oriented Programming (AOP) [15]. Some works advocate that aspects can be used to facilitate PLA evolution by using aspects to modularize PLA variabilities [2], [8], [21]. Aspectual-level components can be used to implement crosscutting non-kernel features, by defining pointcuts that advise base-level components of a PLA. However, recent studies [20], [8] have identified that the use of conventional AOP mechanisms can lead to PLA instabilities in specific evolution scenarios. The reason is twofold: (i) the use of conventional AOP leads to a high coupling between aspectual and base components of a PLA, thereby generating pointcut instabilities, and (ii) many decisions on architectural variation points are still difficult to modularize with AOP.

Emerging AOP approaches, such as XPIs [13], address the first problem but not the second one. XPIs aim at decreasing the tight dependency caused by AOP. This approach employs explicit abstract interfaces, called *Crosscut Programming Interfaces* (XPIs), to decouple aspects from the base code intercepted by them. A XPI specifies sets of base code points where aspects should be plugged in.

In the context of aspect-oriented PLAs, the employment of XPIs can decouple aspectual-level components from the core architecture, thus improving the PLA modularity. However,

the combined use of components, aspects, and XPIs to design PLAs does not suffice to address the scattered implementation of inter-related decisions at architectural variation points. Components usually have to include in their implementation some additional code in order to support all possible variability decisions of the architectural variation points. In this case, the support implementation of such variability decisions is usually scatteredly implemented over the components. Hence, evolution scenarios on architectural variation points would also imply in changes traversing the related components, thereby decreasing the PLA stability.

We propose the concept of aspect-connectors for improving architecture variability, called *Connector-VPs*. While *Connector-VPs* are employed to bind aspectual components to XPIs of base-level components, they also encapsulating the support implementation for variability decisions from components. The goal of our solution is to design stable PLAs by encapsulating the implementation of such otherwise scattered decisions inside *Connector-VPs*. The employment of *Connector-VPs* avoids changes in architectural variation points from being propagated to the components and their interfaces.

This paper also presents a comparative study to evaluate the positive and negative impact of using *Connector-VPs* to design component-based PLAs. The objective is to quantitatively and qualitatively assess to what extent the use of aspect-connector for architecture variability promotes PLA design stability in the presence of various types of change. In our investigation, we have adopted a component implementation model called COSMOS* [12]. We have focused on eight releases of a SPL called *MobileMedia* [16]. Two alternative implementations of *MobileMedia* product line were involved and compared in our study: (i) one using COSMOS* component model; and (ii) one using COSMOS* combined with the use of *Connector-VPs*. It is worth mentioning, that in both implementations, XPI approach was employed to decouple aspectual-level components from base-level ones. We have employed conventional metrics for change impact [23] and modularity [26] for evaluating the PLA stability of the two implementations. The results pointed out that the application of *Connector-VPs* tends to promote superior PLA resilience than the other approach involved in this study.

The paper is organized as follows: Section II presents some necessary concepts to understand the rest of this paper. Section III presents the novel COSMOS*-VP implementation model, which applies our proposed solution. Section IV describes the empirical study, which provides data for the change impact analysis in Section V and for the modularity analysis in Section VI. Section VII presents some works related to this one, and in Section VIII we draw the conclusions, list some limitations of our study, and plan the future work.

II. BACKGROUND

A. COSMOS* Component Implementation Model

According to Szyperski [24], a software component is an unit of modularity with explicit provided and required inter-

faces. It can also be deployed independently and is subject to composition by third parties. The COSMOS* implementation model is representative of component models because it has all these characteristics. The main benefits of COSMOS* is twofold. First, COSMOS* explicitly represents architectural units, such as components, connectors and configuration, thus providing traceability between the software architecture and the respective source code. Second, COSMOS* is considered a platform-independent model, as it is based on a set of design patterns.

COSMOS* defines five sub-models, which address different perspectives of component-based systems: (i) the specification model specifies the components; (ii) the implementation model explicitly separates the provided and required interfaces from the implementation; (iii) the connector model specifies the link between components using connectors; (iv) composite components model specifies high-granularity components; and (v) system model defines a software component which can be straightforwardly executed.

Figure 1 (a) shows an architectural view of a COSMOS* component called *FavouritesMgr* and Figure 1 (b) shows the detailed design of the same COSMOS* component. COSMOS* components are internally divided in specification (*spec* package) and implementation (*impl* package). The specification is the external view of the component, which is also sub-divided in two parts, one that specifies the provided services (*spec.prov* package) and the other makes dependencies explicit (*spec.req* package). For instance, *IManager* and *IFavourites* interfaces are provided interfaces and *IPersistence* is a required interface. The *impl* package has three mandatory classes: (i) a *ComponentFactory* class, responsible for instantiating the component; (ii) a *Facade* class that realizes provided interfaces, following the *Facade* design pattern [10]; and (iii) a *Manager* class that realizes *IManager* interface and provides meta-information about the component. It is possible to have an optional class called *ObjectFactory*, which aims at reducing coupling between implementation classes within the component. *FavControl*, which supports the implementation of the *IFavourites* interface and requires services from other components via *IPersistence* interface, is example of an auxiliary class.

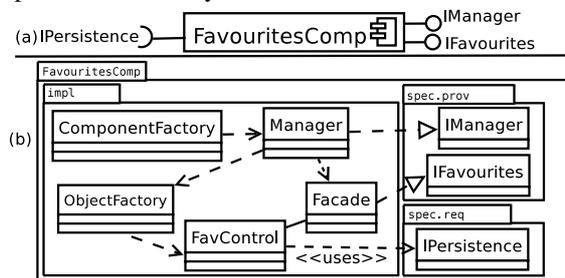


Fig. 1. (a) An architectural view of a COSMOS* component; (b) A detailed design of the COSMOS* component.

B. Components Enriched With XPIs

Some approaches aiming at decreasing the high coupling caused by aspects have been proposed [13], [17]. These

approaches decouple the specification of the base code places where aspects are plugged from the aspects itself, in order to improve system modularity. The XPI approach [13] is one of the proposed approaches. While it employs *Crosscut Programming Interfaces* (XPIs) to specify base code places separately from aspects, it neither limits the use of existing aspect-oriented mechanisms nor require new ones.

In the context of the combined use of components and aspects, we call by aspectual-level components those which implement crosscutting features, and are plugged into base-level components in order to provide their features. Aspectual-level components support the separation of features (in this paper, features are considered equivalent to concerns), and, consequently, increases the system modularity. Applying the XPIs approach to the combined use of components and aspects, the aspects of an aspectual-level component and the XPIs of a base-level components are separated, thus improving system modularity [17].

COSMOS* can combine aspects and the concepts of XPI to the context of components in order to implement aspectual-level components. These components use aspects to intercept base-level components in order to provide their functionality. The main goal is to take advantage of the benefits of the three approaches involved, thus increasing the modularity of component-based architectures. Some characteristics of this combination are similar to other aspectual component models, such as separation between aspectual-level and base-level components, as DyMAC [17] and FAC [22], and the employment of aspect-connectors, as FAC. However, the combined use of COSMOS* and XPI, compared to these approaches, presents some advantages. For example, it does not need new programming mechanisms as the FAC, and its connectors can be used not just to encapsulate non-functional concerns as in DyMAC (see Section VII).

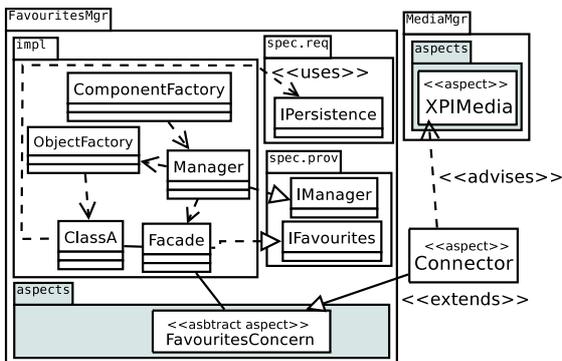


Fig. 2. Detailed design of an aspectual-level component.

Figure 2 shows the detailed design of the *FavouritesMgr* now implemented as an aspectual-level component. It has a structure similar to the COSMOS* component, with a new package called *aspects*. Abstract Aspects that specify advices stay inside the *aspects* package, *FavouritesConcern* is an example of Abstract Aspects. The aspect-connector *Connector* extends *FavouritesConcern* and advises the base-level *MediaMgr*, through its *XPIMedia*. For the sake of clarity,

we omitted all operations, attributes, and the classes inside *MediaMgr* package.

C. Evolving Component-based PLAs with Aspects

Some works advocate that aspects can be used to facilitate PLA evolution by using aspects to modularize PLA variabilities [2], [8], [21]. In that case, aspectual-level components are used to implement non-kernel features. The employment of XPIs can decouple aspectual-level components from base-level ones, thus improving the PLA modularity.

However, the XPIs concept does not suffice to solve the scattering of architectural variation points over architectural elements of a component-based PLA. The scattering is created by implementing an architectural variation point across a set of components. That is usually necessary in order to support all variability decisions related to the architectural variation point.

In an illustrative example, suppose that the PLA of a mobile phone SPL has to handle different types of media, such as music and photo. From the assets of this SPL, it is possible to derive mobile phones which handle (i) music, (ii) photo, or (iii) both music and photo. A possible way to implement this PLA would be creating one component for media, one for music, and other for photo. The *MediaMgr* component would handle operations that are common to both photo and music (e.g. create, delete) and *MusicMgr* and *PhotoMgr* components would handle operations specific to its media type, respectively. Since at least one media type must be chosen, *MediaMgr* is a kernel component implemented on the base level. *MusicMgr* and *PhotoMgr* components are implemented on the aspect level and intercept *MediaMgr* component in order to provide *Music* and *Photo* features, respectively. The *MediaMgr* component is not aware of the particularities of each type of media, since it deals with general operations. Thus, both *PhotoMgr* and *MusicMgr* components must check whether the data originated from *MediaMgr* component is of the appropriate type. This data checking is only necessary when there are more than one type of media, and it represents a support implementation for the decisions of the architectural variation point which is scattered on two components, namely *PhotoMgr* and *MusicMgr*. Hence, evolution scenarios related to this architectural variation point would imply in changes in at least two components.

The support implementations for architectural variation points should be encapsulated within architectural connectors, thus allowing components and architectural variation points to be changed independently. Without such architectural connector, a change in these implementations may affect several components of a PLA. Furthermore, a component that implements variation points of a certain PLA might hinder its reusability in other PLAs, since it holds details of the PLA.

The encapsulation of support implementations for architectural variation points in specific architectural connectors also helps to isolate the implementation of features from the implementation of variation points. Thus, changing one should not affect the other. For instance, when an optional

feature becomes an alternative feature, only the element where the decisions related to this feature are supported should be modified, that is, the specific architectural connector which implements the architectural variation point.

III. COSMOS*-VP EXTENSION

The COSMOS*-VP model extends COSMOS* model by providing guidelines to specify aspect-connectors for architectural variation points, called *Connector-VPs* and materialize them into source code. *Connector-VPs* avoid architectural variation points to be scatteredly implemented over several PLA architectural elements. Once the architectural variation points are moved from components to *Connector-VPs*, changes in architectural variation points are avoided from being propagated to the components and their interfaces, thus facilitating PLA evolution.

TABLE I
SUMMARY OF THE ELEMENTS INTRODUCED TO THE COSMOS*
CONNECTOR MODEL

Element	Description
Delegation Interfaces	Aspects used to extend Abstract Aspects of aspectual-level components.
Interception Interfaces	Aspects used to advise base-level components in order to provide the non-kernel features of the aspectual-level components.
Adapter	A class which implement an Adapter design pattern [10] between the Interception Interfaces and the Delegation Interfaces of a <i>Connector-VP</i> .

Table I summarises the elements introduced by COSMOS*-VP to the COSMOS* connector model, in order to allow the specification and implementation of *Connector-VPs*. A *Connector-VP* provides mechanisms to mediate the binding of the Abstract Aspects of aspectual-level components to the XPIs of base-level components. This binding is necessary in order to provide the non-kernel features of aspectual-level components to base-level components. *ConnectorVPs* bind the components by using *Delegation Interfaces* to extend the Abstract Aspects, and using *Interception Interfaces* to advise the base-level components XPIs. The use of such interfaces separates the *Connector-VPs* specification from its implementation, avoiding a *Connector-VP* from being an instable hard-wired connector between Abstract Aspects and XPIs.

The Adapter design pattern is used to mediate the connection between *Delegation Interfaces* and *Interception Interfaces* (see Table I). While the Adapter mediates the connection, it provides a place to implement the necessary support for variability decisions of the architectural variation point. Hence, the employment of the Adapter helps a *ConnectorVP* to encapsulate an architectural variation point. That is, without a mediator, as an Adapter of a *ConnectorVP*, the base-level components are directly advised by aspectual-level ones, thus, the implementation that supports the variability decisions have to be scatteredly implemented inside components. Another benefit of the Adapter is that some mismatches between Abstract Aspects and XPIs, connected by a *Connector-VP* can be adapted.

It is worth mentioning that all aspectual-level components connections at a specific architectural variation point must be mediated by only one *Connector-VP*. That allows the *Connector-VP* to encapsulate the support implementation for all possible decisions of the point.

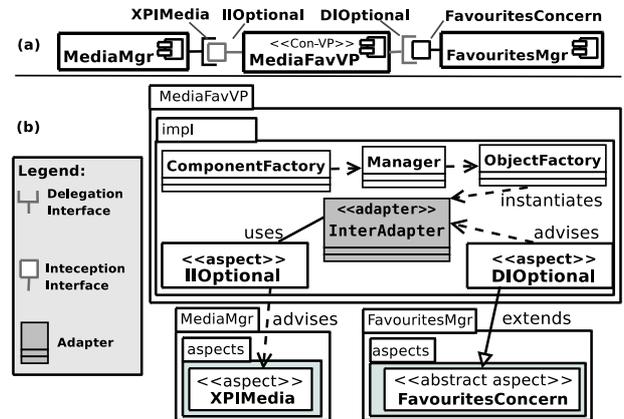


Fig. 3. (a) An architectural view of a *Connector-VP*; (b) A detailed design of a *Connector-VP*.

Figure 3 (a) illustrates the architectural view of a *Connector-VP*, namely *MediaFavVP*, which mediates the connection between the aspectual-level component *FavouritesMgr* to the base-level component *MediaMgr*. The Delegation Interface *DIOptional* extends the Abstract Aspect of *FavouritesMgr*, called *FavouritesConcern*, and the Interception Interface *IIOptional* is used to intercept *MediaMgr*, using *XPIMedia*, in order to provide the optional feature of *FavouritesMgr*.

Figure 3 (b) presents how the *MediaFavVP* can be implemented using AspectJ. In the figure, we have omitted some required packages of the components, for the sake of clarity. To provide the behaviour extended from *FavouritesConcern* to the *MediaMgr* an adapter, called *InterAdapter*, is created between the *IIOptional* and *DIOptional* interfaces. *InterAdapter* must have one method for each one of the advices extended by *DIOptional*. And, each advice of *DIOptional*, which is a realization of one Abstract advice of *FavouritesConcern*, is implemented in order to intercept its correspondent method of the *InterAdapter*. *IIOptional*, during the execution of its advices, which intercept *MediaMgr*, will provide the optional feature *Favourites* to *MediaMgr* calling the *InterAdapter* methods.

This mechanism permits support decisions implementations to be put between the components inside the *MediaFavVP*. These implementations can be created in *IIOptional* or in *InterAdapter*, thus, avoiding them to be scattered over the components. These implementations can determine, for example, under which conditions the optional feature will be provided.

It is important to notice that although the complexity of the *Connector-VPs*, their elements are very simple, and most of them can be semi-automatically generated

based on the XPIs and Abstract Aspects connected by the Connector-VP.

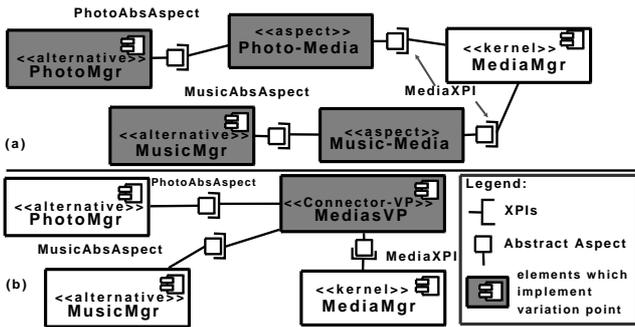


Fig. 4. (a) MobileMedia COSMOS*-XPI PLA; (b) MobileMedia COSMOS*-VP PLA.

Figure 4 illustrates how an architectural variation point can be isolated inside a Connector-VP. The Figure 4 (a) shows a slice of MobileMedia PLA, implemented using COSMOS* combined with XPIs, containing an architectural variation point related to the choice of two alternative components, namely PhotoMgr and MusicMgr. Alternative components are aspectual-level components which implement the alternative features. MusicMgr and PhotoMgr components are implemented on the aspect-level and intercept MediaMgr component in order to provide *Music* and *Photo* features, respectively. The MediaMgr is a base-level component and it deals with general operations of medias support. Both PhotoMgr and MusicMgr component must check whether the data received advising MediaMgr component is of appropriate type. This data checking is not necessary when just one of the components is chosen, and it represents a support implementation for the possible decisions of the architectural variation point, which is scattered over the components.

The Figure 4 (b) shows the same slice of the PLA implemented using COSMOS*-VP. A Connector-VP, called MediasVP, mediates the binding of the alternative components to the MediaMgr. The data checking is implemented into the MediasVP adapter, and depending on the checking result, the appropriated alternative component will provide its feature. Thereby, the architectural variation point was encapsulated only inside the MediasVP.

IV. EMPIRICAL SETTINGS

This section presents the empirical settings used to assess the use of COSMOS*-VP to design stable PLAs.

A. Target software product line

In order to exemplify and evaluate our solution, we present a software application, called MobileMedia [8], which is a SPL for mobile applications that manipulates photo, music, and video on mobile devices, such as mobile phones. The system uses various technologies based on the Java ME platform, such as SMS, WMA and MMAPI. It has two implementations with the same functionalities but implemented with different approaches: one uses AO programming and has approximately 12 KLOC and the other uses only OO programming and has

11 KLOC. MobileMedia endured seven evolution scenarios, which led to eight releases. It is possible to derive 200 products from the last release. The scenarios comprise different types of changes involving kernel, optional, and alternative features, as well as non-functional concerns. The purpose of these changes is to exercise the implementation boundaries and, thus, assess the design stability of the PLA. Table II summarises the evolution scenarios in MobileMedia.

B. Study Definition and Execution

The objective of this comparative study is to assess quantitatively and qualitatively to what extent the specification and implementation of Connector-VPs, by using COSMOS*-VP model, promote PLA design stability in the presence of various types of changes. In this study, we compare two models to implement PLAs: (i) COSMOS* combined with AOP and XPIs, called COSMOS*-XPI; and (ii) using COSMOS*-VP. It is worth mentioning that COSMOS*-VP also employs AOP and XPIs to decouple components from aspectual-level to those of base-level of a PLA. In our study, we have used change impact and modularity metrics in order to evaluate PLAs stability.

The original AO implementation of the MobileMedia was the input for our empirical study. In order to execute the comparative study, we have performed the following steps:

- *Step 1.* Refactor the first release (R1) of the original AO implementation to COSMOS*-XPI and COSMOS*-VP implementations.
- *Step 2.* Evolve the refactored COSMOS*-XPI and COSMOS*-VP implementations according to the evolution scenarios described in Table II.
- *Step 3.* Collect change impact and modularity metrics for eight COSMOS*-XPI and COSMOS*-VP releases;
- *Step 4.* Compare the results of COSMOS*-VP against COSMOS*-XPI implementation.

As a result of *Step 1* execution, two new implementations were created, named COSMOS*-XPI and COSMOS*-VP implementations, each one with eight releases (R1-R8). During the execution of *steps 1* and *2*, we strictly followed the same implementation decisions made by the original MobileMedia developers, such as extracting exception handling code according to Castor et al. [9], and aspectizing all optional and alternative features. During the execution of *Step 3*, we have used the same metric suites of the original MobileMedia empirical study [8]. The majority of the metrics were collected using tools, such as Aopmetrics [1].

V. CHANGE IMPACT ANALYSIS

This section describes the change impact on PLA elements. The change impact on PLA elements is measured by the number of components and connectors changed or added. The greater the number of architectural elements affected (i. e. changed or added), the greater is the impact on the PLA. A PLA is resilient if its elements are little impacted by evolutions. The change impact metrics has been collected comparing each release to its previous one (e.g. comparing R2

TABLE II
SUMMARY OF EVOLUTION SCENARIOS IN MOBILEMEDIA

Release	Description	Type of Change
R1	MobilePhoto core	
R2	Exception handling included (exception handling was implemented according to Castor et al.[9])	Inclusion of non-functional concern which is also a kernel feature
R3	New feature added to count the number of times a photo has been viewed and sorting photos by highest viewing frequency. New feature added to edit the photo's label.	Inclusion of optional and kernel features
R4	New feature added to allow users to specify and view their favourite photos.	Inclusion of optional feature
R5	New feature added to allow users to keep multiple copies of photos	Inclusion of optional feature
R6	New feature added to send photo to other users by SMS	Inclusion of optional feature
R7	New feature added to store, play, and organise music. The management of photo (e.g. create, delete and label) was turned into an alternative feature. All extended functionalities (e.g. sorting, favourites and SMS transfer) were also provided	Changing of one kernel feature into two alternatives
R8	New feature added to manage videos	Inclusion of alternative feature

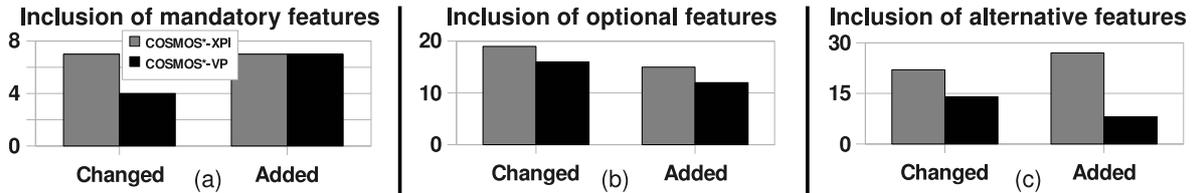


Fig. 5. Number of affected modules during PLA evolution.

to R1). It is worth mentioning that components could also be removed (not only added or changed), but the impact caused by this type of modification was similar for all PLAs and was insignificant.

In the following, we discuss the change impact caused by inclusion of kernel features (R2 and R3), inclusion of optional features (R4, R5, and R6), and inclusion of alternative features (R7 and R8).

A. Inclusion of Kernel Features

This section presents the results of the change impact caused during the inclusion of the kernel features *ExceptionHandler* and *LabelMedia*, which were included in R2 and R3, respectively. The overall results, presented in Figure 5(a), show that COSMOS*-VP PLA had the lowest number of PLA elements changed. That happened because COSMOS*-VP succeeded in isolating changes inside base-level components and inside *Connector-VPs*. Hence, the changes were not propagated to the aspectual-level components connected by the *Connector-VPs*. For example, in R3, the meta-information about media was changed from *String* to *ImageData* type, which implied changes in various XPIs which were relying on that reference. *ImageData* also embodies the same information previously contained in *String* type. On COSMOS*-VP PLA, this type mismatch could be adapted by the *Connector-VPs*. Thus, these changes were isolated in base-level components by the *Connector-VPs*. That was not possible in COSMOS*-XPI PLA. Due to the use of hard-wired connectors, which do not have adapter capabilities, the changes had to be propagated to the aspectual-level components.

In the number of PLA elements added, the result was the same in both PLAs. That happened because kernel features were implemented by base-level components in both PLAs. As COSMOS*-VP does not differ from COSMOS*-XPI on how kernel features are implemented by base-level elements, the same base-level components and connectors needed to be added in both PLAs. Furthermore, no new aspect-connector

was needed. Thereby, the same number of PLA elements was added in both PLAs.

B. Inclusion of Optional Features

This section describes the inclusion of the optional features *Favourites*, *CopyMedia*, and *SMS* (R4, R5, and R6). During the inclusion of these features, COSMOS*-VP PLA presented the lowest number of PLA elements changed and added (see Figure 5 (b)). The results of COSMOS*-VP were possible because of two reasons. First, *Connector-VPs* could be used to connect more than one optional component, added during the evolutions to the aspectual level of the PLAs, to base-level ones. A *Connector-VP* can be reused to connect more than one aspectual-level component, since the components connected are associated with the same architectural variation point. That is not possible with COSMOS*-XPI aspect-connectors, which led to a higher number of added connectors in COSMOS*-XPI PLA. Second, due to the scattered implementation of architectural variation points over COSMOS*-XPI PLA elements, each inclusion of new optional component changed more elements in COSMOS*-XPI than in COSMOS*-VP PLA.

C. Inclusion of Alternative Features

The last two releases (R7 and R8) included the alternative features *Music* and *Video*, respectively. The overall results show a great advantage for COSMOS*-VP against COSMOS*-XPI (see Figure 5 (c)). COSMOS*-VP presented a much lower number of PLA elements added and changed. The employment of *Connector-VP* facilitated the COSMOS*-VP PLA evolution by isolating from components the implementation that supports the variability decisions of the architectural variation points.

In R7, which included the alternative feature *Music*, the kernel feature *Photo* was turned into alternative. In R8 the alternative feature *Video* was also included. These evolution

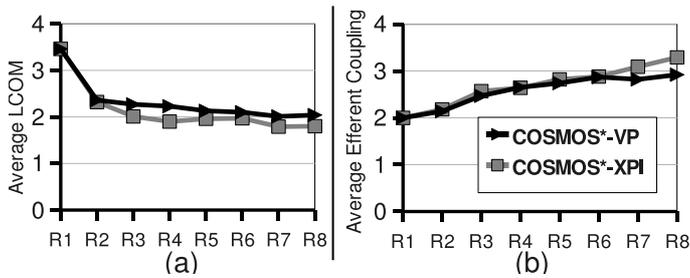


Fig. 6. Coupling and cohesion of MobileMedia PLAs.

scenarios led to a big impact on most of the architectural variation points of the PLAs. They were impacted due to the necessity of considering the new *Music* and *Video* components created. The architectural variation points, which were associated with just optional components, had changed in order to allow different combinations of the optional components and the new alternative ones (*Photo*, *Music*, and *Video*). On COSMOS*-XPI PLA, the optional components were changed in order to create new *Abstract Aspects* to be connected to the new alternative components. And also new aspect-connectors between these components had to be created. On COSMOS*-VP, only the *Connectors-VPs*, already used to connect the optional components to the PLA core, had to be changed to provide the optional features to the new alternative components included. Therefore, the use of *Connector-VP* decreased the number of elements changed and added in these releases, providing the best result to the addition of alternative features (R7 and R8), and also to turn a kernel feature into alternative (R7).

VI. MODULARITY ANALYSIS

This section presents the results for the modularity analysis according to two metrics, namely cohesion and coupling. These metrics were chosen because they are previously-validated stability indicators as presented in several experimental studies (e.g. [8], [9], [11]). The majority of these metrics can be automatically collected by applying metric tools, such as *Aopmetrics* [1].

The modularity of all PLAs is discussed using *Lack of Cohesion of Methods (LCOM)* [6] and *Efferent Coupling (Ce)* [6] metrics. The notion of cohesion is related to encapsulation, that is keeping related things together. Thus, a high LCOM may indicate bad design. Efferent coupling refers to the degree of interdependence between parts of a design, which means that a high interdependence can harm maintainability.

Figure 6 presents PLA modularity in terms of average LCOM and average Ce of all PLA elements measured in each release. The overall results show that COSMOS*-VP presents a PLA implementation as modular as COSMOS*-XPI. It means that the employment of *Connector-VPs* to avoid scattered implementation of architectural variation points did not harm the modular PLA design provided by combined use of components, AOP and XPIs approach.

Regarding LCOM (Figure 6 (a)), COSMOS*-VP PLA presents slightly higher LCOM than COSMOS*-XPI. That happened because the COSMOS*-XPI aspect-connectors are

very cohesive, once they are made by just one aspect bridging XPIs and *Abstract Aspects*. Although, COSMOS*-VP *Connectors-VPs* are simple, they are more complex than COSMOS*-XPI aspect-connectors, thereby they have higher LCOM.

Figure 6 (b) illustrates the results for Ce. COSMOS*-XPI and COSMOS*-VP have very similar overall results for Ce, with a slightly advantages for COSMOS*-VP on last two releases (R7 and R8). The inclusion of alternative features in these releases implied the creation of new aspect-connectors on COSMOS*-XPI PLA. As explained in Section V-C, these connectors, with their dependencies, were not created in COSMOS*-VP PLA.

VII. RELATED WORK

Mezini and Osterman [19] propose a new model called *Caesar*, which allows multiple different decompositions simultaneously. *Caesar* comprises the concept of collaboration interfaces, which differ from standard interfaces in two ways: (i) collaboration interfaces introduce modifiers to annotate required and provided operations; (ii) it uses interface nesting in order to express the interplay between multiple abstractions of a component. Different from *Caesar*, *Cosmos-VP* defines an implementation model which does not demand new mechanisms and can be implemented in mainstream programming languages. Furthermore, *Caesar* does not comprise the concept of aspect-connectors, which is a key concept in COSMOS*-VP model.

Some works propose the integration of components and aspects into new models promoting the encapsulation of advice code (e.g. [17], [22]). Like COSMOS*-VP, one of their goals is to increase reusability of advice code. *FAC* approach [22] also comprises new programming mechanisms and an architecture description language (ADL) to support their approach. *Lagaisse and Joosen* [17] describe a framework whose functional layer contains the core application and the middleware layer offers non-functional services. Aspect-connectors link both layers, which allows non-functional crosscutting concerns to be separately encapsulated. COSMOS*-VP approach differs these by using connectors to encapsulate variation points.

The problem caused by the scattering of architectural variation point is related to optional feature problem, which occurs when optional features are mutual independent in the domain, but have mutual dependencies in their implementation [14]. This problem limits the variability of a PLA. *Kästner et al.* [14] survey different approaches to solve the problem and suggest that derivative modules and conditional compilation can eliminate implementation dependencies and thus restore PLA variability. However, both of these approaches have their shortcomings. Conditional compilation may harm separation of concerns and modularity [8]. *Derivative modules* approach states that code responsible for the dependency should be extracted from the features implementation modules and reimplemented as a new module. Nevertheless, having several modules implementing a feature can harm the stability of the PLA, since whether a evolution in the feature is required,

it may impact on all these modules. `Connector-VP` while mediates interactions among non-kernel features implementation, it avoid these features to be scatteredly implemented over several PLA elements.

Lahire et al. [18] extend the SmartAdapters approach to support variability. The SmartAdapters approach specifies how a reusable concern should be composed with other concerns through a set of adaptations, which are described using a domain-specific language. The extended approach also provides supports for variability in the weaving process in order to make the concern more reusable. Whereas they use adapters to increase the reusability of concerns, our solution employs connectors to encapsulate architectural variation points aiming at increasing PLA stability.

VIII. CONCLUSION AND FUTURE WORK

PLA is a key artefact to achieve a controlled evolution and, hence, it is important for organisations to understand how PLAs evolve and which approaches better support PLA stability. The main contribution of this paper is a novel component implementation model, namely `COSMOS*-VP`, which improves the stability of component-based PLAs, by avoiding scattered implementation of architectural variation points over several PLA elements. To achieve more stable PLAs, our proposed solution encapsulates the implementation needed to support all variability decision related to architectural variation points into aspect-connectors, namely `Connector-VP`. Which are also employed to mediate the connection between aspectual-level components and base-level components.

`COSMOS*-VP` was compared in the presence of heterogeneous evolutionary scenarios against the combined use of components, aspects and XPIs to implement component-based PLAs. The overall results show that the `COSMOS*-VP` PLA was more resilient than the other approach involved. We concluded that encapsulating the implementation needed to support variability decisions inside aspects-connectors, called `Connector-VPs`, `COSMOS*-VP` reduces change propagation. This was similar in all types of change scenarios involving kernel, optional, and alternative features in MobileMedia.

We identified two main **threats to validity** of our study case: (i) the evolution scenarios might not be representative; and (ii) MobileMedia might not be representative of industrial SPLs. Risk (i) cannot be completely avoided owing to the lack of documentation in the literature about industry representative evolution scenarios in SPL. Moreover, we minimize the risk exercising several evolution scenarios, which involved kernel, and non-kernel optional, and alternative features. Regarding risk (ii), even though MobileMedia is a small SPL, it is heavily-based on industry-strength technologies. Furthermore, it has been extensively used and evaluated in previous research [4], [5], [8]. In fact, we are working on a new case study of a more representative SPL. Although `Connector-VPs` can facilitate PLA evolution and the fact that they can be semi-automatically created, their complexity can harm the code comprehension. Thereby, the new case study must comprise an evaluation of the `Connector-VPs` drawbacks.

ACKNOWLEDGMENT

We would like to thank the anonymous referees for the insightful comments. Marcelo Dias is supported by Fapesp/Brazil under grant 2008/02501-9. Leonardo P. Tizzei is supported by Capes/Brazil under grant 05866/2007. Ceclia M. F. Rubira is partially supported by CNPq/Brazil productivity grant 301446/2006-7. Alessandro Garcia is partially supported by Faperj/Brazil distinguished scientist grant E-26/102.211/2009, by CNPq productivity grant 305526/2009-0, by CNPq Universal project grant 483882/2009-7, and by PUC-Rio productivity grant. And also, this research is partially supported by the Service Oriented Software Development through Product Line Engineering Technology project, which is funded by POSTECH, Phang, South Korea.

REFERENCES

- [1] Aopmetrics - <http://aopmetrics.tigris.org/>. Accessed on October 16, 2009.
- [2] V. Alves, et al. Extracting and evolving mobile games product lines. In *LNCS*, volume 3714/2005, 2005.
- [3] C. Atkinson, et al. *Component-based product line engineering with UML*. Addison-Wesley, Boston, MA, USA, 2002.
- [4] I. A. Bertonecello, M. O. Dias, P. H. S. Brito, and C. M. F. Rubira. Explicit exception handling variability in component-based product line architectures. In *WEH '08: Proc. 4th WEH*, USA, 2008. ACM.
- [5] N. Cacho et al. Eflow: taming exceptional control flows in aspect-oriented programming. In *In Proc. of the 7th AOSD*, 2008. ACM.
- [6] S. Chidamber and C. Kemerer. A metrics suite for oo design. *IEEE TSE*, 20(6):476–493, 1994.
- [7] P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
- [8] E. Figueiredo, et al. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE*, 2008.
- [9] F. C. Filho, et al. Exceptions and aspects: the devil is in the details. In *Proc. of International Symposium on Foundations of Software Engineering (FSE)*, USA, 2006. ACM.
- [10] E. Gamma, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [11] A. Garcia, et al. Modularizing design patterns with aspects: a quantitative study. In *Proc. of 4th AOSD*, USA, 2005. ACM.
- [12] L. A. Gayard, C. M. Rubira, and P. A. Guerra. *COSMOS*: a Component System Model for Software Architectures*. Technical Report IC-08-04, Institute of Computing, University of Campinas, 2008.
- [13] W. G. Griswold, et al. Modular software design with crosscutting interfaces. *IEEE Softw.*, 23(1):51–60, 2006.
- [14] C. Kastner et al. On the impact of the optional feature problem: Analysis and case studies. In *In Proc. of the 13th SPLC*. IEEE Press, 2009.
- [15] G. Kiczales, et al. Aspect-oriented programming. In *Proceedings of ECOOP*, pages 220–242. Springer-Verlag, 1997.
- [16] R. Laddad. *AspectJ in Action*. Manning, 2003.
- [17] B. Lagaisse and W. Joosen. Component-based open middleware supporting aspect-oriented software composition. In *In Proc. of Component-Based Software Engineering*, 2005. pages 139–254, 2005.
- [18] P. Lahire, et al. *Model Driven Engineering Languages and Systems*, Volume 47352007, chapter Introducing Variability into Aspect-Oriented Modeling Approaches, Springer, Berlin, 2007.
- [19] M. Mezini and K. Ostermann. *Reliable Software Technologies Ada-Europe*, chap. Modules for Crosscutting Models. Springer, Berlin, 2003.
- [20] C. Nunes, et al. Comparing stability of implementation techniques for multi-agent system product lines. In *Proc. 13rd CSMR*, Germany, 2009.
- [21] J. Oldevik. Can aspects model product lines? In *EA '08: Proc. of the 2008 AOSD WEA*, USA, 2008. ACM.
- [22] N. Pessemier, et al. A model for developing component-based and aspect-oriented systems. *LNCS*, 4089/2006, 2006.
- [23] C. Sant'anna, et al. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings XVII Brazilian Symposium on Software Engineering*, 2003.
- [24] C. Szyperski. *Component Software*. Addison-Wesley, 2002.
- [25] J. van Gurp, et al. On the notion of variability in software product lines. In *Proc. IEEE/IFIP Conf. on Soft. Architecture (WICSA'01)*, USA, 2001.
- [26] S. Yau and J. Collofello. Design stability measures for software maintenance. *IEEE TSE*, 11(9):849–856, 1985.

A Formal Semantics for Decision-oriented Variability Modeling with DOPLER

Deepak Dhungana
Lero – The Irish Software
Engineering Research Centre
University of Limerick,
Limerick, Ireland
deepak.dhungana@lero.ie

Patrick Heymans
Research Center in Information
Systems Engineering (PReCISE)
University of Namur,
Namur, Belgium
patrick.heyman@fundp.ac.be

Rick Rabiser
Christian Doppler Laboratory
for Automated Software Engineering
Johannes Kepler University
Linz, Austria
rabiser@ase.jku.at

Abstract—Variability models define and document software product lines. They provide the basis for automating the derivation of new products, thereby utilizing the flexibility and adaptability of systems. Numerous approaches for variability modeling have been proposed and applied successfully in industry. In our own research we have been developing a decision-oriented approach to variability modeling (DOPLER) that has been applied in different industrial environments. However, decision-oriented approaches have so far been defined only informally and lack a rigorous definition. This can lead to ambiguities and limits the development of tools. In this paper we define the formal semantics of our decision-oriented variability modeling language DOPLERVML as an example of how decision-oriented variability modeling approaches can be formally defined. We explain how DOPLER models can be used for generating product configurations.

Keywords—Decision-oriented Variability Modeling; DOPLER; Formal Semantics; Software Product Lines.

I. INTRODUCTION

Product line engineering (PLE) aims at lowering the cost and increasing the quality of software development by developing a family of similar and related products [3]. Models are used in PLE to explicitly define the often tacit knowledge about variability and to support the automation of different product line activities. For instance, variability models are used to guide and automate product derivation [17] or system reconfiguration [19]. A wide array of variability modeling techniques and tools have been developed reflecting the background of different researchers, the needs of different industrial contexts, and the kinds of systems under investigation. For example, feature-oriented approaches [5], [13], [14] have been developed and applied successfully. Similarly, decision-oriented approaches have been proposed [1], [2], [15], [18] that guide product derivation by domain experts based on decision models.

We have developed the decision-oriented variability modeling language DOPLERVML as part of the DOPLER¹ tool suite [8] and have been successfully applying the approach in a number of industrial contexts. For example, Siemens VAI² uses the approach to automate component-based software

product line engineering [7]. It is also used in the domain of industrial automation systems to support runtime reconfiguration of IEC 61499 based systems using variability models [9]. Furthermore, we have been exploring the use of our model-based approach to support runtime monitoring and adaptation of service-oriented systems [4]. In the domain of enterprise resource planning DOPLER is used for model-based customization of business systems at runtime [19]. These different case studies demonstrate the utility and usefulness of our approach. However, until now the semantics of the modeling constructs is only implicitly defined as a part of the tools supporting the approach. The implementation and adoption of DOPLERVML preceded its formalization which is not unusual as shown in [10].

A number of new research challenges led us to start working on a more rigorous definition of DOPLERVML. An explicit formal semantics enables their unambiguous interpretation and manipulation. Such capabilities are required in our tool suite as several third-party components interact with DOPLER to automate tasks such as generating product configurations, checking the consistency of models and code, or testing product lines.

This paper is structured as follows: In Section II, we present an informal background of DOPLERVML and introduce an example, to be used through out the paper. Section III presents the formal semantics of the approach. In Section IV, we demonstrate how the formal constructs can be used to devise algorithms to that can generate configurations. Section V presents related work and Section VI concludes the paper with a short discussion on future work.

II. BACKGROUND: DOPLERVML

DOPLERVML supports variability modeling of the problem space (stakeholder needs or desired features), the solution space (the architecture and the components of the technical solution), and traceability between these spaces. *Problem space variability* (also known as *product line variability* [16]) is relevant to the domain and needs to be understandable by domain experts utilizing the model for product derivation. Variability models therefore define the available set of choices and the relationships among these. *Solution space*

¹Decision-Oriented Product Line Engineering for Effective Reuse

²<http://www.industry.siemens.com/metals/en/>

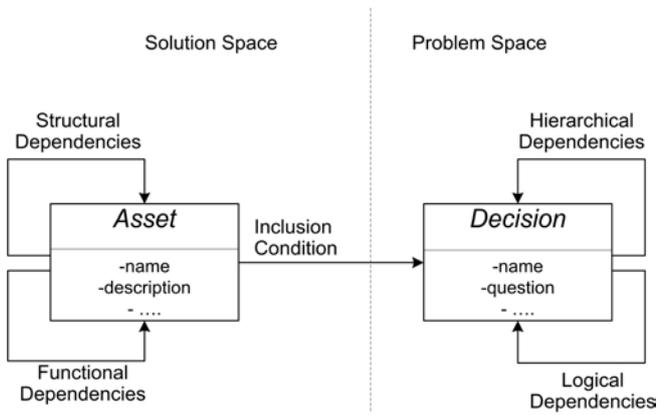


Fig. 1. Core meta-model of the DOPLERVML. Decisions define problem space variability while assets reflect the solution space structure. Inclusion conditions link the problem and the solution space.

variability (also referred to as *software variability* [16]) means the variability of diverse reusable assets such as architectural elements, components, test cases, or documents. Managing variations at different levels of abstraction and for diverse development artifacts is a daunting task, especially when the systems supporting various products are very large, as is common in industrial settings. A language for modeling the solution space needs to be flexible and adaptable to different implementation practices [7].

The core meta-model for DOPLERVML is depicted in Figure 1. It allows building models of the problem space using decisions and models of the solution space using assets. The meta-model is generic and can be adapted to different domains by defining concrete asset types, asset type attributes, and relationships between assets.

A. Decisions

A *decision* is defined whenever for a given goal (e.g., configuring a component) there exist two or more ways of achieving it. Decisions represent the variation points in a product line variability model. Taking a decision involves judging the merits of multiple options and selecting one of them for action (e.g., when considering customer requirements). Decisions are not independent of each other and cannot be made in isolation. For instance, due to decision dependencies earlier decisions can lead to new decisions or influence the options available in subsequent decisions.

Our modeling approach supports two kinds of decision dependencies: firstly, as not all decisions are equally important or relevant at a certain point in time, we allow modeling them in a hierarchy. *Hierarchical dependencies* are used to specify when a particular decision is relevant to a user. The hierarchical arrangement of decisions thus adds context to them. For example, it would make no sense to ask a user about the capacity of a database system, if she does not intend to use a database at all. Secondly, as taking a certain decision may have implications on other decisions we allow modeling decision effects. *Logical dependencies* represent

actions that need to be executed after a decision has been taken to propagate the effect of one decision to other decisions. For example, the type of the database to be used could be logically induced from the system’s size. The core meta-model (Figure 1) shows hierarchical dependencies specifying how the decisions are organized and logical dependencies specifying the relationship between decisions’ values through decision effects.

A simple example of a decision-oriented variability model of a personal information management (PIM) system is depicted in Figure 2. It consists of three decisions describing product line variability and assets describing software variability. When deriving a new product, users first need to decide on the basic functionality provided by the PIM, which is modeled using the decision `PIM_Tasks`. Depending on the chosen option(s) (Email, Todo Lists, Appointments) the user is presented with subsequent decisions (i.e., the decision `Sync` is presented to the user). If the user selects 2-way synchronization (decision `Sync`) she needs to determine the length of the synchronization interval through another decision `Sync_Interval`.

B. Assets

Assets describe the product line’s reusable artifacts and their dependencies relevant in a certain development environment. We use the term *asset* as a generic term to represent all kinds of artifacts whose variability needs to be modeled. This is required, as different mechanisms are typically used to achieve variability at different levels such as requirements, architecture, or implementation. A great challenge lies in linking these variability mechanisms across different artifacts. Using the generic term “asset” thus allows for domain-specific refinements and interpretations that are needed to support different software development processes and environments.

At the meta-level we support two basic types of dependencies among assets: *structural dependencies* are used to specify the physical organization of the assets while *functional dependencies* are used to specify how the system is implemented. *Structural dependencies* describe the physical organization of the assets. This includes how they are packaged or arranged in sub-systems. In our modeling language, structural dependencies are represented by relationship links like “consists of”, “contributes to”, “is predecessor of”, “is successor of”, etc. *Functional dependencies* describe the logical structure of assets and define how they functionally depend on each other. In our modeling language, functional dependencies are represented by relationship links such as “includes”, “requires”, “excludes” etc.

The example model depicted in Figure 2 defines six reusable assets (in this example software components) implementing the PIM application. Models also contain inter-asset relationships reflecting the organization of the solution space. E.g., `POP3` requires a `SSL_Coder` component. In DOPLER such dependencies are modeled formally using a rule language, which will be illustrated later in this paper.

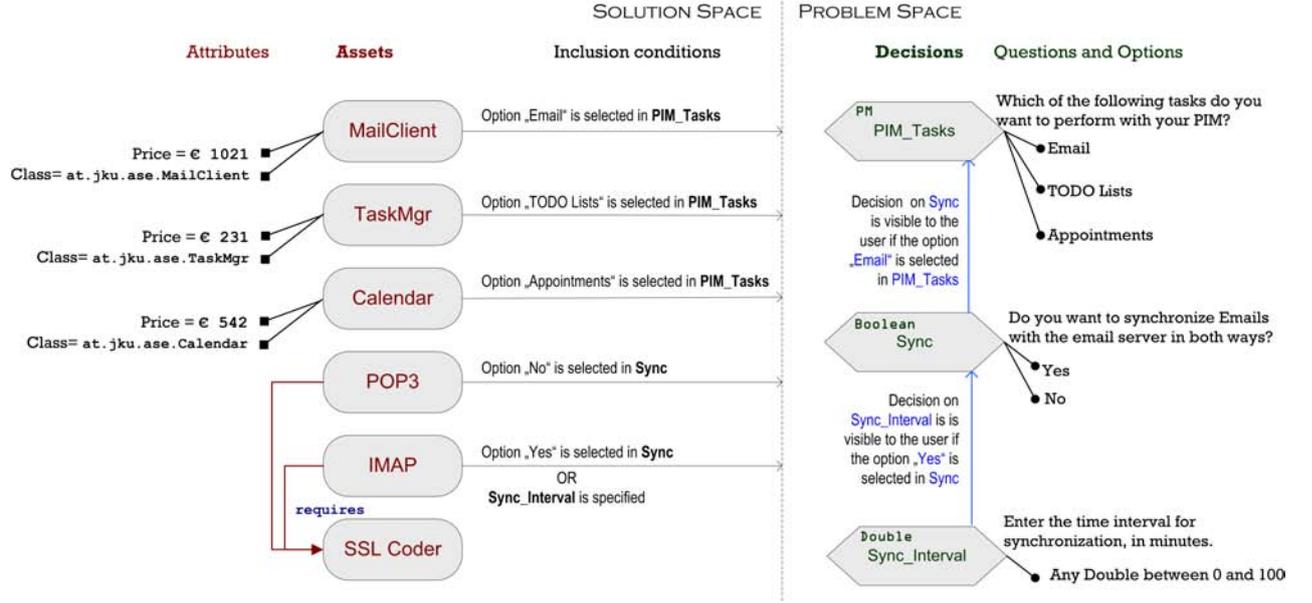


Fig. 2. A simple example of a decision-oriented variability model containing problem space elements (decisions) and solution space elements (assets). For illustrative purposes, the dependencies among the model elements are written in plain text, in DOPLERVML these are modeled using a rule language.

Decisions and Assets are linked using *inclusion conditions*. For example in Figure 2, the components refer to the values of the decisions and specify the conditions under which they are part of the configuration of the derived product. `MailClient` is included if the option `Email` is selected in the decision `PIM_Tasks`. This means that assets are “aware” of the conditions under which they are required for the final product whereas decisions are “unaware” of the assets. This is helpful when marketing strategies of a company change and a new portfolio of products is required for the same set of core assets.

III. FORMAL SEMANTICS

According to [11], [12], the semantics of a language \mathcal{L} (where \mathcal{L} actually denotes its *abstract syntax*) consists of two parts: the *semantic domain*, noted \mathcal{S} , and the *semantic function* $\mathcal{M} : \mathcal{L} \rightarrow \mathcal{S}$. We first describe the building blocks of DOPLERVML and then define its abstract syntax, its semantic domain, and its semantic function.

A. Building Blocks of the Language

Definition 1 (Data type): A data type θ is a couple

$$\theta = \langle \text{Id}_\theta, \text{Dom}_\theta \rangle$$

where Id_θ is the type identifier and Dom_θ is the set of possible values for this type, also called the interpretation domain.

We define \mathcal{DT} to be the set of all types defined in a variability modeling context. Types can be predefined (\mathcal{DT}_p) or user-defined (\mathcal{DT}_u): $\mathcal{DT} = \mathcal{DT}_p \cup \mathcal{DT}_u$.

\mathcal{DT}_p is a finite set of predefined data types:

$$\mathcal{DT}_p = \{ \langle \text{Number}, \mathbb{Q} \rangle, \langle \text{Boolean}, \mathbb{B} \rangle, \langle \text{String}, \mathcal{S} \rangle \}.$$

\mathcal{DT}_u is a finite set of data types to be provided by the modeler. For each user-defined data type θ , we define an

operation $\text{Enum}(\theta)$ which returns the set of values in the Dom_θ . In our example (see Figure 2) the set of user-defined data types is the following singleton whose domain contains an enumeration of three values:

$$\mathcal{DT}_u = \{ \langle \text{PM}, \{ \text{Email}, \text{ToDoList}, \text{Appointment} \} \rangle \}.$$

The interpretation function for types is a function $\llbracket \bullet \rrbracket : \mathcal{DT} \rightarrow \mathcal{Dom}$ which returns the interpretation domain corresponding to the type provided, i.e. $\llbracket \theta \rrbracket \stackrel{\text{def}}{=} \text{Dom}_\theta$. \mathcal{Dom} is the interpretation domain of all data types, i.e. $\mathcal{Dom} = \bigcup_{\theta \in \mathcal{DT}} \text{Dom}_\theta$.

Definition 2 (Boolean Formulae): Boolean Formulae (\mathbb{BF}) play an important role in DOPLERVML. They can be built from terms and simpler sub-formulae. The terms can be constants, variables or expressions built using other terms. Due to the lack of space, we do not give a complete definition. These formulae have a straight forward semantics. As we will see later, the variables in a Boolean formula can also be decisions from the decision model (which are comparable to typed variables).

B. The Abstract Syntax: \mathcal{L}

The abstract syntax for DOPLERVML defines the set of all possible variability models that can be written in the language. A variability model is a couple $\langle \mathcal{DM}, \mathcal{AM} \rangle$ where \mathcal{DM} is a decision model, and \mathcal{AM} is an asset model. \mathcal{DM} and \mathcal{AM} are described in detail in definitions 3 and 4, respectively. For convenience, the inclusion conditions of assets are part of the asset model.

1) *The Decision Model:*

Definition 3 (Decision Model): A decision model \mathcal{DM} can be defined as a 5-tuple

$$\mathcal{DM} = \langle \mathcal{D}, \tau, f_{vis}, f_{val}, f_{pos} \rangle$$

where

- \mathcal{D} is the finite set of decisions provided by the modeler. We differentiate between two kinds of decisions: *user decisions* (\mathcal{UD}), which are directly taken by the user, and *state decision* (\mathcal{SD}), which values are derived from already taken decisions: $\mathcal{D} = \mathcal{UD} \cup \mathcal{SD}$.
- $\tau : \mathcal{D} \rightarrow \mathcal{DT}$ is a typing function labeling each decision with its corresponding data type. Example: $\tau(PIM_Tasks) = \langle PM, \{Email, TodoList, Appointment\} \rangle$ and $\tau(Sync) = \langle Boolean, \mathbb{B} \rangle$. Furthermore, we define a key word *selection*, which returns the current value of a decision. The value that can be bound to decisions always has the same type as the decision itself.
- $f_{vis} : \mathcal{D} \rightarrow \mathbb{BF}$ returns the visibility condition (as a Boolean formula) for each decision. Example: $f_{vis}(Sync) = \{Email\} \in selection(PIM_Tasks)$. The visibility condition of a decision is a type of hierarchical dependency explained in Section II-A.
- $f_{val} : \mathcal{D} \rightarrow \mathbb{BF}$ returns the validity condition for each decision. In the running example, $f_{vis}(Sync_Interval) = selection(Sync_Interval) \geq 0 \wedge selection(Sync_Interval) \leq 100$. The validity condition of a decision is a type of logical dependency between decisions as explained in Section II-A.
- $f_{pos} = f_{der} \cup \tau'$ is a set of rules for deriving the value of decisions. There are two kinds of rules: *value derivation rules* (f_{der}) and *type redefinition* (τ'). These rules are a type of logical dependency between decisions as explained in Section II-A.
- $f_{der} : (\mathcal{SD} \rightarrow \mathbb{BF} \times Terms) \cup (\mathcal{UD} \rightarrow \mathbb{BF} \times Terms)$ is a set of rules defining how the values of certain decisions are calculated depending on whether the specified condition is fulfilled. The definition reflects that all state decisions, but not all user decisions (hence the partial function), have a derivation rule.
- $\tau' \subseteq \mathcal{D} \rightarrow \mathbb{BF} \times \mathcal{DT}$ is a conditional type redefinition function, specifying the condition under which the type of a decision is changed. $\tau'(d) = \langle \varphi, \theta \rangle$ means that if the condition φ is fulfilled, the type of the decision is changed to θ . In the example, $\tau'(PIM_Tasks)$ could be changed to include more options (e.g., *Chat*, *SMS*) depending on how the user chooses other options. For decisions whose type is predefined, type redefinition is forbidden, i.e. $\forall d \in \mathcal{DT}_p. \tau'(d) = \langle TRUE, \tau(d) \rangle$.

Note that there are additional constraints meant to avoid circular definitions. For example, we require that $\forall d \in \mathcal{D}. f_{vis}(d)$ does not refer to d or other decisions influenced by d . The extensive list of constraints is not provided here to save space but they can be found in [6].

2) *The Asset Model:*

Definition 4 (Asset Model): An asset model \mathcal{AM} can be defined as a 9-tuple

$$\mathcal{AM} = \langle \mathcal{A}, \tau, \mathcal{AT}, \mathcal{ATA}, \mathcal{ATR}, f_{av}, R_{inc}, R_{exc}, f_{inc} \rangle$$

where:

- \mathcal{A} is a finite set of assets.
- $\tau : \mathcal{A} \rightarrow \mathcal{AT}$ is the typing function of assets. τ is overloaded wrt \mathcal{DM} without ambiguity.
- \mathcal{AT} is a finite set of asset types defined for a given development context. In the example, there is only one asset type, viz. *Component*.
- \mathcal{ATA} is a finite set of attributes.
- $\mathcal{ATR} \subseteq \mathcal{AT} \times \mathcal{ATA}$ a relation that associates asset types to a set of attributes. Not all asset types need to have attributes, and asset types can have more than one attribute. In the example, asset type *Component* is defined with two attributes: *Price* and *Class*.
- $\tau : \mathcal{ATR} \rightarrow \mathcal{DT} \cup \{\mathbb{E}xpr\}$ is a new (harmless) overloading of the typing function. It associates each asset attribute with a data type. $\mathbb{E}xpr$ is a special given data type used for attributes such that $Dom_{\tau(\mathbb{E}xpr)} = Terms$.
- $f_{av} : \mathcal{A} \times \mathcal{ATR} \rightarrow Dom$ is a function that returns the value of the assets' attributes. The value must be of the type specified in the asset type, i.e.

$$\begin{aligned} \forall \langle at, \alpha \rangle \in \mathcal{ATR}, \forall a \in \mathcal{A}. \tau(a) = at. \\ f_{av}(a, \langle at, \alpha \rangle) \in Dom_{\tau(\langle at, \alpha \rangle)} \end{aligned}$$

For example, the value of the attribute *Price* is of type *Double* and the value of the attribute *Class* is of type *String*.

- $R_{inc} \subseteq \mathcal{A} \times \mathcal{A}$ represents the *inclusion* relationship among the assets. For example, having the asset *POP3* requires to include the asset *SSL coder* as well.
- $R_{exc} \subseteq \mathcal{A} \times \mathcal{A}$ represents the *exclusion* relationship among the assets.
- $f_{inc} : \mathcal{A} \rightarrow \mathbb{BF}$ returns, for each asset, the condition for including it in a configuration based on the value of the decisions.

Again the complete list of constraints on these definitions have been omitted in this paper and can be found in [6].

C. *The Semantic Domain: S*

The semantic domain of DOPLERVML is represented by the set of all configurations that can be derived from all possible variability models.

Definition 5 (Semantic Domain): The semantic domain is defined as $\mathcal{S} = \mathcal{P}(Configuration)$ where *Configuration* is the set of all tuples of the form $\langle DecVals, SelectedAssets, AttrVals \rangle$:

- $DecVals : \mathcal{D} \rightarrow (Dom \cup \{null\}) \times \{vis, hid\}$ returns a tuple for every decision in the decision model. The first element of that tuple is the value of the decision. If the value is *null*, it means that the decision has not yet been taken. The second element of the tuple is either *vis* or

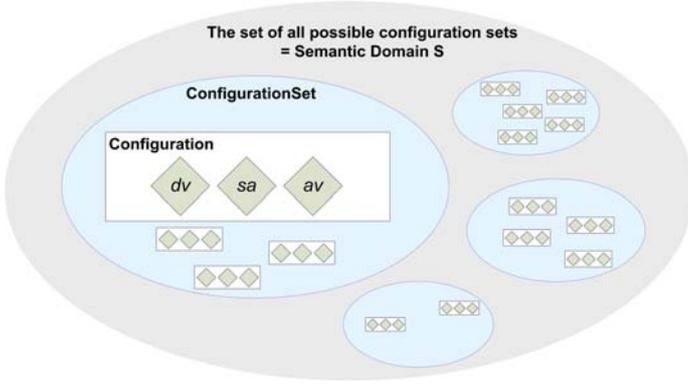


Fig. 3. The semantic domain represents all possible configuration sets. One variability model maps to one particular configuration set, which consists of a possibly infinite number of configurations. Each configuration consists of the set of decision values (dv), the set of selected assets (sa) and the attribute values of the selected assets (av).

hid, indicating that, after the configuration, the decision is visible or hidden, respectively. In a valid configuration c , it is not possible for a decision to be visible but not yet taken:

$$\forall cs \in \mathcal{S}. \forall \langle dv, sa, av \rangle \in cs. dv(x) \neq \{null, vis\}.$$

- $SelectedAssets \subseteq \mathcal{A}$ is the set of assets that have been selected to be included in the final product. There can be two causes for the inclusion of assets:
 - the inclusion condition of the asset evaluates to *true* with the given set of $DecVals$;
 - the asset is the target of a R_{inc} relationship originating from an already included asset.
- $AttrVals \subseteq SelectedAssets \times ATTR \rightarrow Dom$ assigns a value to the attributes of all assets that are included in the final product.

D. The Semantic Function: \mathcal{M}

Given a syntactic domain \mathcal{L} and a semantic domain \mathcal{S} , the final and main step in defining a semantics is to relate the syntactic expressions to the elements of the semantic domain, so that each syntactic creature is mapped to its meaning [12]. The semantic function $\mathcal{M} : \mathcal{L} \rightarrow \mathcal{S}$ for DOPLERVML is defined as

$$\forall m \in \mathcal{L}. \mathcal{M}(m) = \{\langle dv, sa, av \rangle \in \mathcal{S} \mid r1 \wedge r2 \wedge r3\}$$

where $r1$, $r2$ and $r3$ are three rules defined as follows:

- r1*: The values given to decisions meet the specification of their validity, visibility, value derivation and type redefinition³. Formally,

$$\forall d \in \mathcal{D} \cdot \begin{aligned} & \llbracket f_{val}(d) \rrbracket_{dv} \\ & \wedge (\llbracket f_{vis}(d) \rrbracket_{dv} \Leftrightarrow \pi_2(dv(d)) = vis) \\ & \wedge \llbracket f_{der}(d) \rrbracket_{dv} \\ & \wedge \llbracket \tau'(d) \rrbracket_{dv} \end{aligned}$$

³We adopt the usual notation $\pi_i(x)$ for the i^{th} element of a tuple x .

- For a given decision d , validity ($f_{val}(d)$) and visibility ($f_{vis}(d)$) return Boolean formulae that are interpreted in the standard way. Concretely, for f_{val} , this means that the interpretation function $\llbracket \bullet_1 \rrbracket_{\bullet_2} : f_{val} \rightarrow DecVals \rightarrow \mathbb{B}$ is defined as

$$\llbracket d \mapsto \varphi \rrbracket_{dv} \equiv \llbracket \varphi \rrbracket_{dv}$$

The semantics of f_{vis} is similar.

- The interpretation function for derivation rules ($f_{der}(d)$) has a similar signature as above, but gets a slightly different definition:

$$\llbracket d \mapsto \langle \varphi, T \rangle \rrbracket_{dv} \equiv \llbracket \varphi \Rightarrow d = T \rrbracket_{dv}$$

If the condition φ holds, the value of the decision d is calculated by evaluating the term T .

- The interpretation function for type redefinition τ' has, again, a similar signature, but a slightly more subtle interpretation:

$$\begin{aligned} \llbracket d \mapsto \langle \varphi, \theta \rangle \rrbracket_{dv} & \equiv \pi_1(dv(d)) \in \pi_2(\tau(d)) \vee \\ & (\llbracket \varphi \rrbracket_{dv} \wedge \pi_1(dv(d)) \in \pi_2(\theta)) \end{aligned}$$

If the condition φ holds, the value of the decision d is from the new domain defined by the type redefinition function, otherwise the value is from the previously defined domain.

- r2*: The set of selected assets is chosen based on whether the inclusion condition is fulfilled and whether the asset is required by already included assets:

$$sa = sa' \cup \{a \mid \langle b, a \rangle \in R_{inc}^+ \wedge b \in sa'\}$$

where $sa' = \{a \mid a \in \mathcal{A} \wedge \llbracket f_{inc}(a) \rrbracket_{dv}\}$ and where R_{inc}^+ denotes the transitive closure of R_{inc} . We must also avoid explicitly excluded assets to be included, i.e., $sa \cup ua = \emptyset$ where

$$ua = \{a \mid a \in \mathcal{A} \wedge b \in sa \wedge \langle b, a \rangle \in R_{exc}\}$$

- r3*: Finally, an attribute of a selected asset either has a value entered by the modeler, or, if the type of attribute is \mathbb{Expr} , the value returned by the evaluation of the term defined by the attribute value.

$$\begin{aligned} \forall a \in sa. \tau(a) = at \quad \wedge \quad \mu = (a, \langle at, \alpha \rangle) : \\ av(\mu) = \llbracket f_{av}(\mu) \rrbracket_{dv} \quad \vee \quad (\tau(\langle at, \alpha \rangle) \in \mathcal{DT} \\ \wedge av(\mu) = f_{av}(\mu)) \end{aligned}$$

IV. INTERPRETING VARIABILITY MODELS

A DOPLER variability model represents a set of configurations. The “execution” of a variability model selects one of these configurations based on the decisions taken by the user and/or an automated tool. The formal semantics of DOPLERVML allows devising algorithms for automating operations on variability models.

Based on the formal semantics, we have developed tools that can generate all possible configurations based on a variability model. Our configuration generator simulates user interaction

by taking decisions on behalf of the user. We have also developed tools that enact DOPLER variability models by letting users take decisions and arrive at a set of included assets for a specific product configuration. The set of included assets can then be used by domain-specific application generators and deployment tools for further processing.

Firstly, the visibility condition (f_{vis}) of each decision is evaluated. If the condition holds, the decision needs to be taken. Either the user or a tool is asked for an answer. The answer is evaluated against the decision's validity condition (f_{val}). If the validity condition holds, the decision is bound to the input value. The effects of all decisions are propagated by evaluating all derivation rules and executing them as necessary. Such rules can also cause other variable bindings, which lead to the recursive propagation of the decision effects.

The rule engine used to propagate the effects of the decisions does not use a brute force algorithm but evaluates only expressions containing the currently taken decision after changes. The execution of the action can change the set of already bound decision variables; it can however also be only informative. As the rule engine often re-triggers the evaluation of the rules, there must not be cyclic dependencies in the model. We detect cycles statically in the rules using standard graph algorithms.

The set of all included assets are calculated in three steps: (i) An asset is included if its inclusion condition (f_{inc}) holds; (ii) the inclusion relationship of all included assets (R_{inc}) is evaluated and all related assets are also added to the list of included assets; (iii) the second step is repeated until there are no more relationships left to evaluate.

V. RELATED WORK

The first decision-oriented variability modeling approaches were developed in the early 1990's as part of the Synthesis project [2]. Decision-oriented approaches use decisions as prime modeling constructs to define product lines and to support product derivation. Over the years, numerous approaches have been proposed by different researchers and there is abundant work on using them in different contexts [1], [2], [15], [18]. However, decision-oriented approaches so far lack semantic precision which leads to ambiguous interpretations of the models, prevents rigorous analyzes, and makes automation difficult. There are only few tools supporting decision-oriented PLE.

Regarding decision modeling, the DOPLER approach is based on ideas presented in the Synthesis project [2] and the approach by Schmid and John [18]. However, compared to these approaches DOPLERVML provides a more flexible way to model the solution space (because of meta-modelling capabilities) and traceability to the problem space.

Other related decision-oriented approaches include KobrA [1] and V-Manage [15]: The KobrA approach provides no explicit support to model the product line solution space. It is defined only informally and tool support is lacking. The V-Manage approach is tool-supported. However, an explicit formal semantics is so far missing. A unique feature of this

approach is the concept of *decision collections*, i.e., multiple instances of a decision or a set of decisions. For example, when two instances of a certain component are required, the configuration of these components has to be repeated for each required component, i.e., decisions need to be duplicated when executing the variability model.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a formal semantics of the DOPLERVML modeling approach. This semantics is intended to contribute towards a better understanding of decision-oriented variability modeling languages in general. We described how a DOPLER variability model can be interpreted.

DOPLER models are particularly suitable for product derivation scenarios, where stakeholders are presented with a configuration questionnaire. Furthermore, the correctness of the generated configurations depends on the expert knowledge "codified" in the model and not only on the formal semantics described in this paper.

There are other applications of the formal semantics on our agenda. For example, we plan to implement user assistance during modeling to avoid logical modeling errors. We also plan to provide automated analysis of decision interactions based on the assets they influence based on our formal semantics.

REFERENCES

- [1] C. Atkinson, J. Bayer, and D. Muthig. Component-based product line development: the KobrA approach. In *SPLC*, pages 289–310, 2000.
- [2] G. H. Campbell, S. R. Faulk, and D. M. Weiss. Introduction to Synthesis. Technical report, Software Productivity Consortium, Herndon, VA, USA, 1990.
- [3] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering, Addison-Wesley, 2001.
- [4] R. Clotet, D. Dhungana, X. Franch, P. Grünbacher, L. Lopez, J. Marco, and N. Seyff. Dealing with changes in service-oriented computing through integrated goal and variability modeling. In *Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2008)*, pages 43–52, Essen, Germany, 2008. ICB-Research Report No. 22.
- [5] K. Czarnecki and C. Kim. Cardinality-based feature modeling and constraints: A progress report. In *International Workshop on Software Factories at OOPSLA'05*, pages 1–9, San Diego, USA, 2005. ACM Press.
- [6] D. Dhungana. *A Model-driven Approach to Flexible and Adaptable Software Variability Management*. PhD thesis, Johannes Kepler University, 2009.
- [7] D. Dhungana, P. Grünbacher, and R. Rabiser. Domain-specific adaptations of product line variability modeling. In *IFIP WG 8.1 Working Conference on Situational Method Engineering: Fundamentals and Experiences*, Geneva, Switzerland, 2007.
- [8] D. Dhungana, R. Rabiser, P. Grünbacher, and T. Neumayer. Integrated tool support for software product line engineering. In *Tool Demonstration, 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, Atlanta, Georgia, USA, 2007.
- [9] R. Froschauer, A. Zoitl, and P. Grünbacher. Development and adaptation of IEC 61499 automation and control applications with runtime variability models. In *7th IEEE Int'l Conference on Industrial Informatics (INDIN 2009)*, Cardiff, UK, 2009.
- [10] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.
- [11] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff, part I: The basic stuff. Technical report, Jerusalem, Israel, Israel, 2000.
- [12] D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10):64–72, 2004.

- [13] P. Heymans, P. Y. Schobbens, J. C. Trigaux, Y. Bontemps, R. Matulevicius, and A. Classen. Evaluating formal properties of feature diagram languages. *Software, IET*, 2(3):281–302, 2008.
- [14] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.
- [15] J. Mansell and D. Sellier. Decision model and flexible component definition based on XML technology. In *Lecture Notes in Computer Science: Software Product-Family Engineering 5th International Workshop, PFE 2003*, pages 466–472. Springer Berlin/Heidelberg, 2004.
- [16] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *15th IEEE International Requirements Engineering Conference (RE'07)*, pages 243–253, New Delhi, India, 2007.
- [17] R. Rabiser, P. Grünbacher, and D. Dhungana. Supporting product derivation by adapting and augmenting variability models. In *11th International Software Product Line Conference (SPLC 2007)*, Kyoto, Japan, 2007.
- [18] K. Schmid and I. John. A customizable approach to full-life cycle variability management. *Journal of the Science of Computer Programming, Special Issue on Variability Management*, 53(3):259–284, 2004.
- [19] R. Wolfinger, S. Reiter, D. Dhungana, P. Grünbacher, and H. Prähofer. Supporting runtime system adaptation through product line engineering and plug-in techniques. In *7th IEEE International Conference on Composition-Based Software Systems (ICCBSS)*, Madrid, Spain, 2008. IEEE Computer Society.

A deontic logical framework for modelling product families

Patrizia Asirelli, Maurice H. ter Beek, Stefania Gnesi
Istituto di Scienza e Tecnologie dell'Informazione
ISTI-CNR, Pisa, Italy
Email: {asirelli,terbeek,gnesi}@isti.cnr.it

Alessandro Fantechi
DSI, University of Florence, Italy
ISTI-CNR, Pisa, Italy
Email: fantechi@dsi.unifi.it

Abstract—We discuss the application of deontic logics to the modelling of variabilities in product family descriptions. Deontic logics make it possible to express concepts like permission and obligation, and hence promise a direct modelling of constraints over the products of a family. Indeed, we first show how feature models can be straightforwardly characterised by means of a deontic logic. We then study the deontic modelling of the behavioural variability in product families by defining a deontic extension of a behavioural logic. This allows both constraints over the products of a family and constraints over their behaviour to be expressed in a single framework: a novelty in the field. We discuss how model-checking tools could support formal verification in this framework, and we indicate some future research into that direction.

I. INTRODUCTION

Modelling variability in product families has been the subject of extensive study in the literature on Software Product Lines, especially that concerning Feature modelling [3], [8], [15]. Variability modelling addresses how to define which features or components of a system are optional, alternative, or mandatory; formal methods are then developed to show that a product belongs to a family, or to derive instead a product from a family, by means of a proper selection of the features or components.

Modal Transition Systems (MTSs) have been proposed as a formal model for product families [12], [18], allowing one to embed in a single model the behaviour of a family of products that share the basic structure of states and transitions, transitions which can moreover be seen as mandatory or possible for the products of the family. In [10], we have pushed the MTS concept to a more general form, allowing more precise modelling of the different kinds of variability that can typically be found in the definition of a product family.

Recently, deontic logics [1], [22] have become popular in computer science for formalising descriptive and behavioural aspects of systems. This is mainly because they provide a natural way to formalise concepts like violation, obligation, permission, and prohibition. Intuitively, they permit one to distinguish between correct (normative) states and actions on the one hand and non-compliant states and actions on the other hand. This makes deontic logics a natural candidate for expressing the conformance

of members of a family of products with respect to variability rules.

Such a conformance of products to a family concerns not only properties related to features, that in some sense can be considered static. Behavioural variability of the family has to be considered as well, i.e. how the products of a family differ in their ability to respond to events in time. These dynamic properties must also be verified for products to be member of such families. This is an aspect that the techniques focussing on feature models do not typically address.

Recently, a Propositional Deontic Logic (PDL) capable of expressing the permitted behaviour of a system has been proposed [5], [6]. This PDL combines the expression of permission and obligation with concepts from temporal logics.

In [2], we have laid the basis for the study of the application of deontic logics to the modelling of behavioural variability. We did this by showing the capability of a logic derived from PDL to finitely characterise the complete behaviour of a family of products. We have also shown in [2] that, given an MTS \mathcal{M} representing a family, we are able to produce a deontic logic formula that is satisfied by all and only those products that can be derived from \mathcal{M} . This preliminary result has convinced us that this kind of deontic logics are a good candidate to express in a unique framework both behavioural variability aspects, by means of standard branching-time logical operators, and static constraints over the products of a family, which usually require a separate expression in a first-order logic (as can be seen in [3], [11], [19]), using deontic operators.

In the first part of this paper, we present a straightforward characterization of feature models by means of deontic logics. We then proceed with this direction of research by defining a novel deontic extension of a behavioural logic that allows both static constraints over the products of a family and constraints over their behaviour to be expressed in a single framework. This logic is given a semantic interpretation over MTSs for which a verification framework based on model-checking techniques could be implemented extending existing model checking tools.

A. Related Work

MTSs have been introduced in [12], [17], [18] to formally model and verify the behaviour of product families. We have extended MTSs in [10] to allow different notions

Funded by the Italian project D-ASAP (MIUR-PRIN 2007) and by the RSTL project XXL of the Italian National Research Council (CNR).

of behavioural variability to be modelled. An algebraic approach to behavioural modelling and verification of software product lines has been developed in [13], [14]. In [2] we showed how to finitely characterise MTSs by means of deontic logic formulae. To the best of our knowledge, the current paper presents a first attempt towards a modelling and verification framework capable of addressing both static and behavioural conformance of products of a family.

B. Outline

In Sect. II we present a simple running example that we will use throughout the paper. After a brief description of feature models in Sect. III, we discuss the use of deontic logic to characterise feature models and to verify static requirements of product families in Sect. IV. We then introduce the behavioural modelling of families by means of MTSs in Sect. V. In Sect. VI we introduce a deontic extension of an existing branching-time logic, which we apply to the running example in Sect. VII to show its expressivity and to discuss the verification of behavioural requirements of product families in Sect. VIII. We conclude the paper in Sect. IX with some ideas for future work.

II. RUNNING EXAMPLE: COFFEE MACHINE FAMILY

To illustrate the contribution of this paper we use a simple running example, namely a family of (simplified) coffee machines, for which we consider the following requirements:

- 1) A coffee machine is activated by a coin. The only accepted coins are the one euro coin (1€), exclusively for the European products and the one dollar coin (1\$), exclusively for the US products;
- 2) After inserting a coin, the user has to choose whether or not (s)he wants sugar, by pressing one of two buttons, after which (s)he may select a beverage;
- 3) The choice of beverages (coffee, tea, cappuccino) varies between the products. However, delivering coffee is a must for every product of the family, while cappuccino is only offered by European products;
- 4) After delivering the appropriate beverage, optionally, a ringtone is rung. However, a ringtone must be rung whenever a cappuccino is delivered;
- 5) The machine returns to its idle state when the cup is taken by the user.

This list contains both static requirements, which identify the features that constitute the different products (see requirements 1, 3 and, partially, 4) and behavioural requirements, which describe the admitted sequences of operations (requirements 2, 5 and, partially, 4).

In the sequel, we will first distill the feature model of the above family and provide a formal representation in terms of deontic logic formulae. We will then show how the behavioural requirements of this family can be described using an MTS. Finally, we will show how to combine the two approaches by defining a deontic logical

framework to check the satisfiability of both static and behavioural requirements over products that should belong to the family.

III. FEATURE DIAGRAMS AND FEATURE MODELS

Feature diagrams were introduced in [15] as a graphical *and/or* hierarchy of features; the features are represented as the nodes of a tree, with the product family being the root. Features come in several flavours; in this paper we consider the following features:

optional features	may be present in a product only if their parent is present;
mandatory features	are present in a product if and only if their parent is present;
alternative features	are a set of features among which one and only one is present in a product if their parent is present.

When additional constraints are added to a feature diagram, this results in a *feature model*. Also constraints come in several flavours; in this paper we consider the following constraints:

requires	is a unidirectional relation between two features indicating that the presence of one feature requires the presence of the other;
excludes	is a bidirectional relation between two features indicating that the presence of either feature is incompatible with the presence of the other.

An example of a feature model for the Coffee Machine of Sect. II is given in Fig. 1; the **requires** constraint obligates feature Ringtone to be present whenever Cappuccino is, while the **excludes** constraint prohibits features 1\$ and Cappuccino to both be present in any product of this family of Coffee Machines. It is easy to see that this feature model satisfies the static requirements (1, 3 and, part of, 4) of our running example.

IV. DEONTIC LOGIC APPLIED TO FEATURE MODELS

Deontic logics are an active field of research for many years now. Many different deontic logic systems have been developed with a lot of success in the community [1], [22].

A. Deontic Logic: A First Glimpse

A deontic logic consists of the standard operators of propositional logic (i.e. negation (\neg), conjunction (\wedge), disjunction (\vee) and implication (\implies)) augmented with deontic operators. In this paper, we consider two of the most classic deontic operators, namely *it is obligatory that* (O) and *it is permitted that* (P), which enjoy the duality property

$$P(\alpha) = \neg O(\neg\alpha),$$

i.e. something is permitted iff its negation is not obligatory.

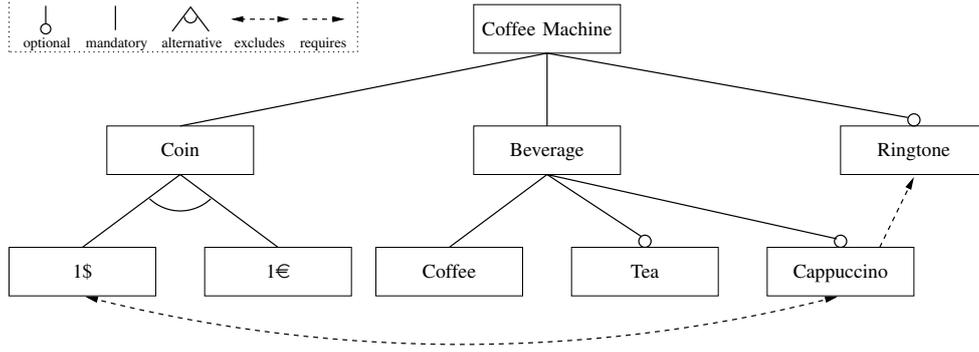


Figure 1. Feature model of the Coffee Machine family.

B. A Deontic Characterization of Feature Models

The way deontic logics formalise concepts such as violation, obligation, permission and prohibition is very useful for system specification, where these concepts arise naturally. In particular, deontic logics seem to be very useful to formalise product families specifications, since they allow one to capture the notions of optional and mandatory features.

The deontic characterization of a feature model builds a set of deontic formulae which, taken as a conjunction, precisely characterises a product family. We assume that a name of a feature A is used as the atomic proposition indicating that A is present.

The deontic characterization is constructed as follows:

- If A is a feature, and A_1 and A_2 are two subfeatures (possibly marked **alternative**, **optional** or **mandatory**), then add the formula

$$A \implies \Phi(A_1, A_2),$$

where $\Phi(A_1, A_2)$ is defined as:

$$\Phi(A_1, A_2) = (O(A_1) \vee O(A_2)) \wedge \neg(P(A_1) \wedge P(A_2))$$

if A_1 and A_2 are marked **alternative**, and is otherwise defined as:

$$\Phi(A_1, A_2) = \phi(A_1) \wedge \phi(A_2),$$

in which A_i , for $i \in \{1, 2\}$, is defined as:

$$\phi(A_i) = \begin{cases} P(A_i) & \text{if } A_i \text{ is } \mathbf{optional} \text{ and} \\ O(A_i) & \text{if } A_i \text{ is } \mathbf{mandatory}. \end{cases}$$

Moreover, since the presence of the root feature is taken for granted, the premise of the implication related to that feature can be removed.¹

- If A **requires** B , then add the formula

$$A \implies O(B)$$

- If A **excludes** B , then add the formula

$$(A \implies \neg P(B)) \wedge (B \implies \neg P(A))$$

Before applying this construction to our running example, we make two remarks. First, note that the **alternative**

¹By doing so, we tacitly do not deal with trivially inconsistent graphs in which the root is involved in an excludes relation with a feature.

feature can also be defined in terms of the **excludes** feature: Marking subfeatures A_1 and A_2 **alternative** is the same as stating that A_1 and A_2 **exclude** each other.

The second remark concerns the following less common feature: **multiple optional** features are a set of features of which at least m and at most $n > m$, with $m, n \geq 0$, are present in a product if their parent is present. The deontic characterization of this feature is as follows, assuming that feature A has s subfeatures A_1, \dots, A_s :

$$A \implies \left(\bigvee_K \left(\bigwedge_{i \in K} O(A_i) \wedge \bigwedge_{j \notin K} \neg P(A_j) \right) \right),$$

with $K = \{ S \subseteq \{1, \dots, s\} : m \leq |S| \leq n \}$.

If we apply the construction described above to our running example, then the conjunction of the following deontic formulae precisely characterises the feature model of Fig. 1. We refer to this conjunction as the *characteristic formula*.

$$O(\text{Coin}) \wedge O(\text{Beverage}) \wedge P(\text{Ringtone})$$

$$\text{Coin} \implies (O(1\$) \vee O(1\text{€})) \wedge \neg(P(1\$) \wedge P(1\text{€}))$$

$$\text{Beverage} \implies O(\text{Coffee}) \wedge P(\text{Tea}) \wedge P(\text{Cappuccino})$$

$$\text{Cappuccino} \implies O(\text{Ringtone})$$

$$(1\$ \implies \neg P(\text{Cappuccino})) \wedge (\text{Cappuccino} \implies \neg P(1\$))$$

C. Verifying Static Requirements of Product Families

The deontic characterization of feature models described in this section is a way to provide a semantics to feature models. The deontic formulae expressing the features and the constraints between them can then be used to verify whether or not a certain product belongs to a specific family.

Given the above characterization of the coffee machine, let us suppose that we have defined two coffee machines with the following list of features:

$$\text{CM1} = \{\text{Coin}, 1\text{€}, \text{Beverage}, \text{Coffee}\}$$

$$\text{CM2} = \{\text{Coin}, 1\text{€}, \text{Beverage}, \text{Coffee}, \text{Cappuccino}\}$$

It is easy to see that coffee machine CM1 belongs to the product family since it satisfies the characteristic formula of the feature model, whereas CM2 does not: it falsifies the constraint that a Cappuccino requires a Ringtone. This can be formally verified by interpreting these lists as a conjunction of axioms (each comma stands for a \wedge) that when added to the characteristic formula makes it either true or false, according to whether or not the product belongs to the family. For instance, CM2 does not belong to the product family because the addition (through conjunction) of

$$\text{Coin} \wedge 1\text{€} \wedge \text{Beverage} \wedge \text{Coffee} \wedge \text{Cappuccino}$$

to the characteristic formula of the feature model makes the subformula $\text{Cappuccino} \implies O(\text{Ringtone})$ false, as a result of which the whole formula is false.

In general, the problem of finding a product that satisfies the deontic characterization of a feature model is reduced to that of finding a satisfying assignment to a set of boolean variables. Efficient SAT solvers, like Chaff [21], can therefore be used to address this kind of problems.

V. BEHAVIOURAL MODELS FOR PRODUCT FAMILIES

The behavioural requirements given in Sect. II for our coffee machine family can be formally expressed by a Modal Transition System (MTS). Several variants of MTSs have been proposed in [10], [12], [18] with the aim of embedding in a single model the behaviour of a family of products that share the basic structure of states and transitions. This basic structure can be defined as a doubly-labelled transition system [9], which is an extension of an ordinary Labelled Transition System (LTS) obtained by labelling states with atomic propositions and transitions with actions.

Definition 1: A *doubly-labelled transition system* (L^2TS) is a sextuple $(S, s_0, Act, \rightarrow, AP, L)$, in which

- S is a set of states;
- $s_0 \in S$ is the initial state;
- Act is a finite set of observable actions;
- $\rightarrow \subseteq S \times Act \times S$ is the transition relation; instead of $(s, \alpha, s') \in \rightarrow$ we will often write $s \xrightarrow{\alpha} s'$;
- AP is a set of atomic propositions;
- $L : S \rightarrow 2^{AP}$ is a labelling function that associates a subset of AP to each state.

An MTS can now be defined as an L^2TS in which transitions are either required or possible, to reflect mandatory or optional transitions for the products of the family.

Definition 2: A *modal transition system* (MTS) is a septuple $(S, s_0, Act, \rightarrow_{\square}, \rightarrow_{\diamond}, AP, L)$ such that $(S, s_0, Act, \rightarrow_{\square} \cup \rightarrow_{\diamond}, AP, L)$ is an L^2TS . A MTS has two distinct transition relations: The *must* relation \rightarrow_{\square} expresses *required* transitions, while the *may* relation \rightarrow_{\diamond} expresses *possible* transitions. Note that, by definition, the may relation includes the must transition.

An example MTS is shown in Fig. 2; the solid arcs are must transitions, while the dashed arcs are may transitions. Its states are $\{0, 1, \dots, 12\}$, with initial state 0, while its set of actions contains 1€ , $1\text{\$}$, sugar , no_sugar , etc.

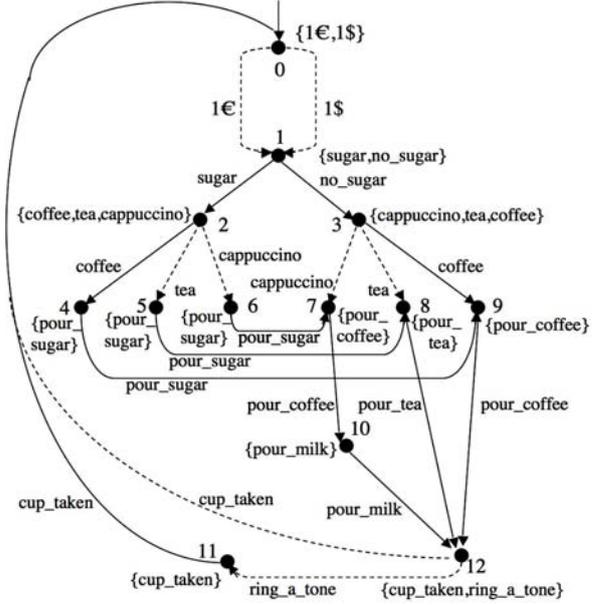


Figure 2. MTS modelling a product family.

Each state is labelled with the set of actions that label its outgoing (required and possible) transitions. Finally, must transitions $2 \xrightarrow{\text{coffee}}_{\square} 4$ and $3 \xrightarrow{\text{coffee}}_{\square} 9$ imply that delivering coffee is a must for every product of the family represented by this MTS. In fact, this MTS models the behavioural requirements given in Sect. II for our family of coffee machines.

Note that the MTS in Fig. 2 also models the static requirements concerning **optional** and **mandatory** features, through the use of may and must transitions. However, this MTS is not able to model three particular constraints listed among the requirements:

- actions 1€ and $1\text{\$}$ are exclusive (**alternative** features);
- a cappuccino is only offered by European products (**excludes** relation between features);
- a ringtone is rung whenever a cappuccino is delivered (**requires** relation between features).

We have seen that deontic logics can express also these characteristics, as represented by the Feature Diagram of Fig. 1. In order to define a unique framework in which to reason about behavioural as well as static requirements, in the following section we work on the integration of deontic operators within a temporal logic able to deal with MTSs.

VI. TOWARDS A DEONTIC LOGICAL FRAMEWORK FOR PRODUCT FAMILIES

In order to define a unique logical framework in which to express both evolution in time and the variability of a product family, we define the temporal logic DHML based on the “Hennessy-Milner logic with Until” defined in [9], [16], which has been augmented with the Deontic *possibility* and *obligation* operators (in a style reminiscent of the logic PDL proposed in [5], [6]) and path operators from CTL [7]. DHML is a simpler variant of the logic proposed in [2].

A. DHML Logic: Syntax

DHML is a logic of state formulae (denoted by ϕ), in which a path quantifier prefixes an arbitrary path formula (denoted by π). We assume a set of atomic actions $Act = \{\alpha, \beta, \dots\}$ and a set of atomic propositions $AP = \{p, q, \dots\}$. From these two sets more complicated formulae can be built in the usual way, using the propositional and deontic operators described in Sect. IV or actions as well as the Hennessy-Milner modal, CTL path, and Until operators we describe next, together with their intuitive meaning:

- $[a]\phi$: for all next states reachable with a , ϕ holds;
- $E\pi$: there exists a path on which π holds;
- $A\pi$: on each of the possible paths π holds;
- $\phi U \phi'$: in the current or a future state ϕ' holds, while ϕ holds until that state (but not necessarily in that state).

Definition 3: The syntax of DHML is:

$$\begin{aligned} \phi &::= tt \mid p \mid \neg\phi \mid \phi \wedge \phi' \mid [\alpha]\phi \mid E\pi \mid A\pi \mid \\ &\quad P(\alpha) \mid O(\alpha) \\ \pi &::= \phi U \phi' \end{aligned}$$

As usual, ff abbreviates $\neg tt$, $\phi \vee \phi'$ abbreviates $\neg(\neg\phi \wedge \neg\phi')$, $\phi \implies \phi'$ abbreviates $\neg\phi \vee \phi'$, and $\langle\alpha\rangle\phi$ abbreviates $\neg[\alpha]\neg\phi$: there exists a next state reachable with a in which ϕ holds. Furthermore, $F\phi$ abbreviates $(tt U \phi)$: there exists a future state in which ϕ holds; and $G\phi$ abbreviates $\neg F(\neg\phi)$: in any future state ϕ holds. Finally, $EF\phi$ abbreviates $E(tt U \phi)$: there exists a path on which ϕ holds in a future state; and $AG\phi$ abbreviates $\neg EF\neg\phi$: ϕ holds in every state on every path.

An example of a well-formed formula in DHML is thus

$$[\alpha](P(\beta) \wedge (p \implies O(\gamma))),$$

which states that after the execution of the action α , the system is in a state where the action β is *permitted* (in the sense of the *may* transition) and if the proposition p holds then the action γ is *obligatory* (in the sense of the *must* transition).

B. DHML Logic: Semantics

The formal semantics of DHML is given below by means of an interpretation over the MTSs defined in Sect. V. To this purpose, we use a relation $P \subseteq S \times Act$ to denote which actions are permitted in which states, with the understanding that $P(s, \alpha)$ iff $\alpha \in L(s)$. We assume that $Act \subseteq AP$, i.e. all actions are atomic propositions.

Definition 4 (Semantics): The satisfaction relation \models of DHML over an MTS $\mathcal{L} = (S, s_0, Act, \rightarrow, \rightarrow_\square, \rightarrow_\diamond, AP \cup Act, L)$ is defined as follows:

- $s \models tt$ always holds;
- $s \models p$ iff $p \in L(s)$;
- $s \models \neg\phi$ iff not $s \models \phi$;
- $s \models \phi \wedge \phi'$ iff $s \models \phi$ and $s \models \phi'$;
- $s \models [\alpha]\phi$ iff $s \xrightarrow{\alpha}_\square s'$, for some $s' \in S$, implies $s' \models \phi$;

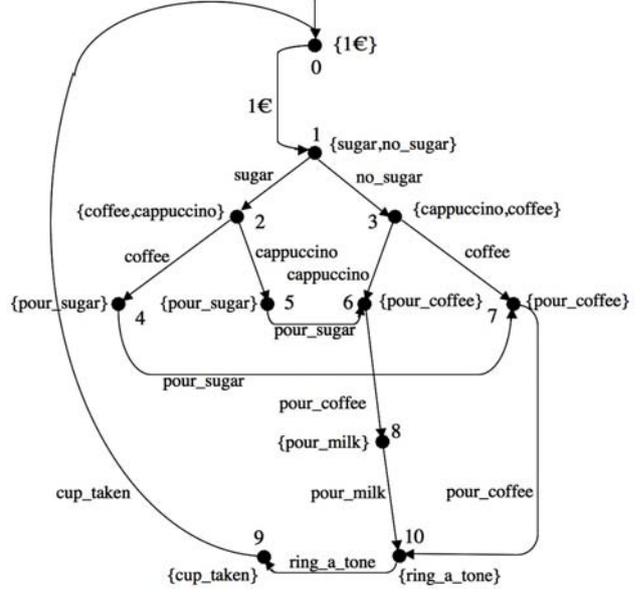


Figure 3. MTS of a European Coffee Machine.

- $s \models E\pi$ iff there exists a path σ starting in state s such that $\sigma \models \pi$;
- $s \models A\pi$ iff $\sigma \models \pi$ for all paths σ starting in state s ;
- $s \models P(\alpha)$ iff $P(s, \alpha)$ holds;
- $s \models O(\alpha)$ iff $P(s, \alpha)$ holds and $\exists s' : s \xrightarrow{\alpha}_\square s'$;
- $\sigma \models [\phi U \phi']$ iff there exists a state s_j , for some $j \geq 0$, on the path σ such that for all states s_k , with $j \leq k$, $s_k \models \phi'$ while for all states s_i , with $0 \leq i < j$, $s_i \models \phi$.

For the MTS in Fig. 2 we thus have, e.g., $0 \models 1\text{€}$ and $0 \models [1\text{€}](O(\text{sugar}))$ since $1 \models (O(\text{sugar}))$, which itself follows from the fact that $\text{sugar} \in L(1)$ and $1 \xrightarrow{\text{sugar}}_\square 2$.

Note that notions of weak and strong permission are introduced in [5], [6] (and used to define a notion of obligation). The semantics of DHML can be extended in the following way to include a notion P_w of *weak permission*: $s \models P_w(\alpha)$ iff $P(s, \alpha)$ holds and $\exists s' : s \xrightarrow{\alpha}_\diamond s'$.

Finally, we note that DHML differs from the classical modal μ -calculus [16], since the modal box operator of DHML is defined in terms of may transitions while the modal μ -calculus makes no distinction between must and may transitions in its semantic domain. For the same reason, also the weak permission operator cannot be expressed in the modal μ -calculus.

VII. USING DHML TO EXPRESS BEHAVIOURAL AND STATIC REQUIREMENTS OF PRODUCT FAMILIES

We can now apply the DHML logic introduced in the previous section to our running example. We do this to illustrate the ability of DHML to express both static constraints over the products of a family and constraints over their behaviour.

DHML is able to complement the behavioural description of an MTS by expressing constraints over possible products of a family, that is, the static requirements that could not be expressed in the MTS:

- actions 1€ and 1\$ are exclusive (**alternative** features):

$$\begin{aligned} ((EF \langle 1\$ \rangle true) \implies (AG \neg P(1€))) \wedge \\ ((EF \langle 1€ \rangle true) \implies (AG \neg P(1\$))) \end{aligned}$$

- a cappuccino is only offered by European products (**excludes** relation between features):

$$\begin{aligned} ((EF \langle cappuccino \rangle true) \implies (AG \neg P(1\$))) \wedge \\ ((EF \langle 1\$ \rangle true) \implies (AG \neg P(cappuccino))) \end{aligned}$$

- a ringtone is rung whenever a cappuccino is delivered (**requires** relation between features):

$$(EF \langle Cappuccino \rangle true) \implies (AF O(ring_a_tone))$$

The above expressions have been obtained by merging the static requirements represented by the pure deontic formulae given in Sect.IV for these requirements, with the behavioural relations among actions expressible by the temporal part of the logic. It is worthwhile making two remarks. First, note that we have used the same characterization of the **alternative** feature as the one given for the **excludes** feature.

Second, since **requires** is a static relation between features it does not imply any ordering among the related features, i.e. a coffee machine that rings a ringtone before producing a cappuccino cannot be excluded as a product of the family of coffee machines by verification of the above formula. Indeed, the correct ordering of actions is guaranteed by the MTS description of the family.

DHML is also able to express behavioural requirements over possible products of a family as temporal logic properties, such as:

- 1) It is possible to get a coffee with 1€:

$$[1€] EF \langle coffee \rangle true$$

- 2) It is always possible to ask for sugar:

$$AF \langle sugar \rangle true$$

- 3) It is not possible to get a beverage without inserting a coin:

$$\begin{aligned} AG (\neg (coffee \vee tea \vee cappuccino) U \\ (\langle 1€ \rangle true \vee \langle 1\$ \rangle true)) \end{aligned}$$

It is important to note, however, that the logical concept of *possibility* does not distinguish between the concepts *possibility for a user of the coffee machine to ask for sugar* and *possibility for a product of the family to include, among the other actions offered to the user, the action of asking a cappuccino*. To distinguish these two concepts of possibility, we need to resort to the deontic operators of DHML, using its capability to combine the expression of the concepts of permission and obligation with that of behavioural requirements.

VIII. USING DHML TO VERIFY BEHAVIOURAL AND STATIC REQUIREMENTS OF PRODUCT FAMILIES

Another classic application of temporal logic is to verify that a model of a system satisfies properties given by logic formulae. Model checking is the most known automatic technique for verifying a system's correctness properties [7]. Such verification is exhaustive, i.e. all possible input combinations and states are taken into account, and a counterexample is usually generated in case a certain property does not hold. Correctness properties reflect typical (un)desired behaviour of the system under scrutiny. Formally, the problem of model checking can be stated as follows: given a desired property, expressed as a temporal logic formula ϕ , and a structure \mathcal{M} with initial state 0, decide whether $\mathcal{M}, 0 \models \phi$, where \models is the usual satisfaction relation. If \mathcal{M} is finite, model checking thus reduces to a graph search.

We could use model checking to analyse the conformance of members of a family of products with respect to variability rules. To do so, we consider that a product is formally represented by a MTS in which only must (required) transitions appear. For instance, let us consider the product represented by the coffee machine defined by the MTS presented in Fig. 3. Such a MTS may have been generated starting from an independent high-level specification language such as, e.g., UML, and we may want to check that it belongs to the family, by checking the properties that we have defined to characterise the coffee machine family. It is easy to check that all the properties previously defined are satisfied by this MTS.

Moreover, if we take a few examples of properties expressed in DHML that are a mix of behavioural and deontic characteristics, then we are interested in checking their validity over the MTS presented in Fig. 3. If they turn out to be valid, then we can conclude that this product satisfies all the static (features) and behavioural requirements that the products derived from the family of coffee machines should satisfy according to the list of requirements given in Sect. II.

A first and simple example is the formula

$$EF O(coffee),$$

which must be read as *it is possible that eventually the product is obligated to deliver a coffee*, i.e. *there exists a sequence of actions that leads to a state in which there is a must transition labelled coffee*. Verifying this formula on this model of a product shows it is valid, because in state 2, e.g., there is a must transition labelled *coffee* ($2 \xrightarrow{\text{coffee}} \square 4$) and $coffee \in L(2)$.

Note that the presence of a may transition labelled *cappuccino* has no influence on the verification of this formula: To be valid in a state s , the obligation $O(coffee)$ requires s to be labelled with *coffee* and the presence of an outgoing must transition from s labelled with *coffee*.

It is immediate that this formula implies the validity of

$$EF P(coffee).$$

Finally, since all paths at a certain point pass either state 2 or 3 and $\text{coffee} \in L(3)$ and $3 \xrightarrow{\text{coffee}} \square 7$, even the formula

$$AF O(\text{coffee})$$

is valid: always eventually a coffee must be delivered.

In general, to perform verifications of this kind, we need a model checker able to check DHML formulae over models described as MTSs, with possible constraints expressed in DHML itself.

We are currently pursuing two different approaches to DHML model checking:

- We can exploit the relations between MTSs and L^2 TSSs in order to reuse the UMC model-checking engine [20]. UMC is an on-the-fly model checker that was originally designed for the efficient verification of UCTL logic [4], an action- and state-based branching-time temporal logic, over L^2 TSSs. The comparison of the expressiveness of UCTL and DHML still has to be studied, which means that enhancements to the model-checking engine to cover DHML deontic operators could be needed as well.
- Several model checkers employ SAT-solvers to implement the so-called *bounded model checking* approach, in order to efficiently address large state spaces. Using the same SAT-based engine for solving both the deontic issues related to the constraints on a family (as seen before) and the behavioural issues (employing bounded model-checking techniques) is hence a way of pursuing the scalability of verification of DHML properties to real-world cases, in which state spaces tend to increase beyond the capability of explicit model checkers.

Merging the two approaches with the aim of increased scalability and usability would then be a further step in the direction of the industrial application of the verification of behavioural requirements of product families.

IX. CONCLUSIONS AND FUTURE WORK

In [2] we have shown how a deontic logic can express the variability of a product family by showing the capability of a deontic logic formula to finitely characterise a finite state MTS, a formal method proposed to capture the behavioural variability of a product family. In this paper, we have pursued this line of research. We have first shown how feature models can be straightforwardly characterised by means of a deontic logic. We have then defined DHML, a novel deontic extension of a Hennessy-Milner and CTL-like behavioural logic for product families that allows both static constraints over the products of a family and constraints over their behaviour to be expressed in a single framework. The semantic domain of this logic has been chosen such that a verification framework based on model-checking techniques is available.

The added value of the DHML logic we have introduced in this paper can thus be summarised as allowing the possibility of reasoning, in a unique framework, also on behavioural aspects of products of a family.

There are several aspects of our line of research that require a deeper understanding:

- how to express dependencies of variation points;
- the identification of classes of properties that, proved on family definitions, are preserved by all the products of the family;
- how quantitative properties can be evaluated, such as the number of possible different products of a given family;
- from a more pragmatic point of view, the study on scalability to real problems, and how the approach adapts to incremental family construction.

More importantly, it remains to study to what degree the complexity of the proposed deontic logic and verification framework can be hidden from the end user, or can be made more user friendly, in order to support developers in practice, since formal models such as MTSs are not directly usable as modelling framework. Nowadays, UML diagrams are often used as modelling paradigm and it could be very interesting to be able to apply to them the same formal reasoning we have presented here for MTSs and the deontic logic. Indeed, recently model-checking techniques for UML activity and state chart diagrams have been developed [4], exploiting the branching-time temporal logic UCTL. An extension of this framework by including deontic operators could be applied to verify the conformance of static and dynamic constraints of product derivations. This would allow to go into the direction of producing the family description itself already in a UML-like fashion, hence towards a better usability and acceptance within the industrial software product lines community.

REFERENCES

- [1] L. Åqvist, Deontic Logic. In D. Gabbay and F. Guenther (Eds.): *Handbook of Philosophical Logic* (2nd Edition), Volume 8. Kluwer Academic, 2002, 147-264.
- [2] P. Asirelli, M.H. ter Beek, A. Fantechi, and S. Gnesi, Deontic Logics for Modeling Behavioural Variability. In D. Benavides, A. Metzger, and U. Eisenecker (Eds.): *Proceedings of the Third International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'09)*, ICB Research Report 29, Universität Duisburg-Essen, 2009, 71-76.
- [3] D.S. Batory, Feature Models, Grammars, and Propositional Formulas. In J.H. Obbink and K. Pohl (Eds.): *Proceedings Software Product Lines Conference (SPLC'05)*, LNCS 3714, 2005, 7-20.
- [4] M.H. ter Beek, A. Fantechi, S. Gnesi and F. Mazzanti, An action/state-based model-checking approach for the analysis of communication protocols for Service-Oriented Applications. In S. Leue and P. Merino (Eds.): *Proceedings Formal Methods for Industrial Critical Systems (FMICS'07)*, LNCS 4916, Springer, 2008, 133-148.
- [5] P.F. Castro and T.S.E. Maibaum, A Complete and Compact Propositional Deontic Logic. In C.B. Jones, Zh. Liu and J. Woodcock (Eds.): *International Colloquium Theoretical Aspects of Computing (ICTAC'07)*, LNCS 4711, Springer, 2007, 109-123.

- [6] P.F. Castro and T.S.E. Maibaum, A Tableaux System for Deontic Action Logic. In R. van der Meyden and L. van der Torre (Eds.): *Proceedings Deontic Logic in Computer Science (DEON'08)*, LNCS 5076, Springer, 2008, 34–48.
- [7] E.M. Clarke, E.A. Emerson, and A.P. Sistla, Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions of Programming Languages and Systems* 8, 2 (1986), 244–263.
- [8] K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.
- [9] R. De Nicola and F.W. Vaandrager, Three Logics for Branching Bisimulation. *Journal of the ACM* 42, 2 (1995), 458–487.
- [10] A. Fantechi and S. Gnesi, Formal Modeling for Product Families Engineering. In *Proceedings Software Product Lines Conference (SPLC'08)*, IEEE, 2008, 193–202.
- [11] A. Fantechi, S. Gnesi, G. Lami and E. Nesti, A Methodology for the Derivation and Verification of Use Cases for Product Lines. In R.L. Nord (Ed.): *Proceedings Software Product Lines Conference (SPLC'04)*, LNCS 3154, Springer, 2004, 255–265.
- [12] D. Fischbein, S. Uchitel and V.A. Braberman, A Foundation for Behavioural Conformance in Software Product Line Architectures. In R.M. Hierons and H. Muccini (Eds.): *Proceedings Role of Software Architecture for Testing and Analysis (ROSATEA'06)*, ACM, 2006, 39–48.
- [13] A. Gruler, M. Leucker and K.D. Scheidemann, Modeling and Model Checking Software Product Lines. In G. Barthe and F.S. de Boer (Eds.): *Proceedings Formal Methods for Open Object-Based Distributed Systems (FMOODS'08)*, LNCS 5051, Springer, 2008, 113–131.
- [14] A. Gruler, M. Leucker and K.D. Scheidemann, Calculating and Modeling Common Parts of Software Product Lines. In: *Proceedings Software Product Lines Conference (SPLC'08)*, IEEE, 2008, 203–212.
- [15] K. Kang, S. Choen, J. Hess, W. Novak and S. Peterson, Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical Report SEI-90-TR-21, Carnegie Mellon University, Nov. 1990.
- [16] K.G. Larsen, Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion, *Theoretical Computer Science* 72, 2-3, (1990), 265–288
- [17] K.G. Larsen and B. Thomsen, Partial Specifications and Compositional Verification, *Theoretical Computer Science* 88, 1, (1991), 15–32
- [18] K.G. Larsen, U. Nyman and A. Wąsowski, Modal I/O Automata for Interface and Product Line Theories. In R. De Nicola (Ed.): *Proceedings European Symposium on Programming Languages and Systems (ESOP'07)*, LNCS 4421, Springer, 2007, 64–79.
- [19] M. Mannion and J. Camara, Theorem Proving for Product Line Model Verification. In F. van der Linden (Ed.): *Proceedings Software Product-Family Engineering (PFE'03)*, LNCS 3014, Springer, 2004, 211–224.
- [20] F. Mazzanti, UMC model checker v3.6, April 2009. URL: <http://fmt.isti.cnr.it/umc>
- [21] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang and S. Malik, Chaff: Engineering an Efficient SAT Solver. In: *Proceedings Design Automation Conference (DAC'01)*, ACM, 2001, 530–535.
- [22] J.-J.Ch. Meyer and R.J. Wieringa (Eds.), *Deontic Logic in Computer Science: Normative System Specification*, Wiley, 1994.

The Variability Model of The Linux Kernel

Steven She*, Rafael Lotufo*, Thorsten Berger†*, Andrzej Wasowski‡, Krzysztof Czarnecki*

*University of Waterloo, Canada, {shshe, rlotufo, kczarnec}@gsd.uwaterloo.ca

†University of Leipzig, Germany, berger@informatik.uni-leipzig.de

‡IT University of Copenhagen, Denmark, wasowski@itu.dk

Abstract—Lack of realistic benchmarks hinders efficient design and evaluation of analysis techniques for feature models. We extract a variability model from the code base of the Linux kernel, obtaining a model larger by an order of magnitude than the largest publicly available feature model so far. We analyze properties of this model, compare it with previously available benchmarks, and emphasize the differences from published academic examples. As a result, we broaden our understanding of what a feature model is, hopefully challenging tool designers by providing an interesting benchmark, giving input to design of random model generators, and last but not least, inspiring designers of variability modeling languages.

I. INTRODUCTION

Reports from tool vendors and users in the series of proceedings of the Software Products Lines conference witness a broad industrial interest and experience in using variability modeling. Nevertheless, many researchers feel that realistic benchmarks for evaluating variability modeling tools are inaccessible [1]. Many variability models *are* in fact available already (see www.splot-research.org, fm.gsdlab.org), however very few of them originate from realistic processes; most are small examples from research publications or outcomes of student run case studies. Given the lack of realistic large-scale models, many authors resort to using randomly generated models [2], [3], [4].

In order to help the community mitigate this limitation, we bring a large and realistic variability model, extracted from the build system of the Linux kernel. Linux has an explicit variability specification expressed in the Kconfig language [5]—a language developed specifically for this purpose by the kernel developers. The Kconfig model so closely resembles feature models [6], [7] that it can be naturally interpreted as one [8].

We present the Kconfig model—available online in a feature modeling friendly format (see fm.gsdlab.org). We detail the model transformation from the Kconfig language to feature model. Foremost, we extensively study the properties of the Kconfig model, contrasting it to metrics of a corpus of published feature models.

We intend to attract the attention of designers of feature modeling tools to our result, so they can use the Kconfig model as a particularly tough benchmark and its characteristics as a source of requirements on tools. Some would also be interested to know that randomly generated models used in previous works are likely much easier to analyze than the Linux model. Last but not the least, the Kconfig notation, together with the size and structure of the Linux model, should inspire designers of variability modeling languages, in particular when

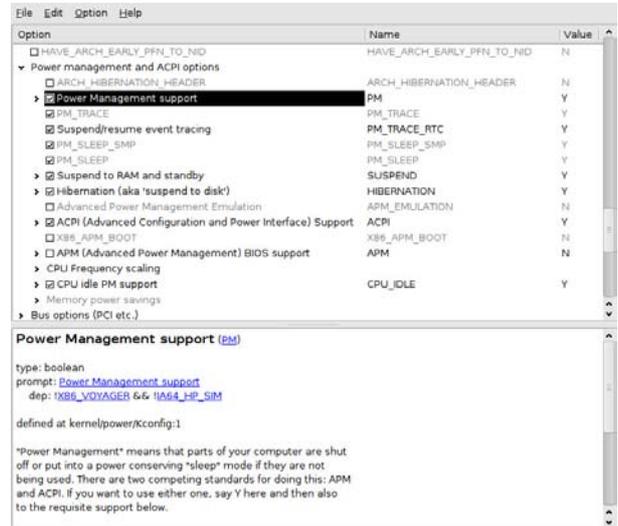


Fig. 1. The xconfig configurator GUI

it comes to support for modularity and user interface aspects like visibility of features.

II. THE KCONFIG LANGUAGE

Configuration options are known as *configs* in Linux. Kconfig is the language used to specify the available configs and dependencies among them: configs can be nested under other configs; they can also be grouped under *menus* and *choice groups*. The kernel configurator, xconfig, renders the Kconfig model as a tree of options, which users select to specify configurations to be built. Cross-tree dependencies, if any, are indicated in the bottom part of xconfig's GUI (Fig. 1).

Figure 2 shows a fragment of the Linux variability model in the Kconfig language. The fragment contains a menu with four Boolean configs as children.

Configs are named parameters with a specified value type: Boolean, tristate, integer (**int** or **hex**), or string. Boolean configs represent options that can be switched on (y) or off (n). Tristate configs are similar, except that they have two 'on' states: y indicates that the code implementing the option should be linked into the kernel statically, whereas m indicates that it should be compiled as a dynamically loadable module. Thus, tristate is a simple form of a *binding mode* specification [7]. We refer to Boolean and tristate configs collectively as *switch configs*, since they appear in xconfig as switches (e.g., checkboxes). Integer configs are used to specify

```

menu "Power management and ACPI options"
  depends on !X86_VOYAGER
  config PM
    bool "Power Management support"
    depends on !IA64_HP_SIM
    ---help---
      "Power Management" means that ...
  config PM_DEBUG
    bool "Power Management Debug Support"
    depends on PM
  config CPU_IDLE
    bool "CPU idle PM support"
    default ACPI
  config PM_SLEEP
    bool
    depends on SUSPEND || HIBERNATION ||
      XEN_SAVE_RESTORE
    default y
...
endmenu

```

Fig. 2. Fragment of a Kconfig model

numerical options such as buffer sizes. String configs are used to specify names of, for example, files or disk partitions. We refer to integer and string configs as *entry-field configs*, since the configuration tool shows them as editable fields.

Config definitions can include other elements besides type. If the type is followed by a *prompt*, i.e., a short explanation text shown to the user in xconfig, the config is *user-selectable*; otherwise it is not. A config can have a *visibility condition* directly following the prompt (not shown in the example). A longer *help* text can also be provided (see the PM entry in Figure 2).

A *depends-on* clause introduces a dependency that must be satisfied when selecting the config. In the example, PM can only be selected if IA64_HP_SIM is not selected. Reversely, a *select* clause (not shown) enforces immediate selection of another config when this config is selected by the user. Depends-on is also used to specify nesting, when referencing its previous config: for example PM_DEBUG is nested under PM.

A *default* clause has a two-fold effect. First, if the config is user-selectable, *default* is used to provide an initial value, which can still be overridden by the user. For example, CPU_IDLE takes the same value as ACPI by default. If the config is not user-selectable, then the default enforces a value for the feature, effectively defining a cross-tree constraint. For example, PM_SLEEP is non-user-selectable and set to y if its depends-on condition holds; otherwise it is set to n.

Menus are used for grouping. Kconfig provides a conditional visibility mechanism for menus. We call conditionally visible menus simply *conditional menus*; if their condition, a cross-tree constraint, is false, they and their children are grayed out in the configurator.

Finally, *choices* (not shown) group configs into alternatives, which we call *choice configs*. Choices themselves can be Boolean or tristate. When the choice state is y, the choice configs underneath are constrained with XOR; when the value

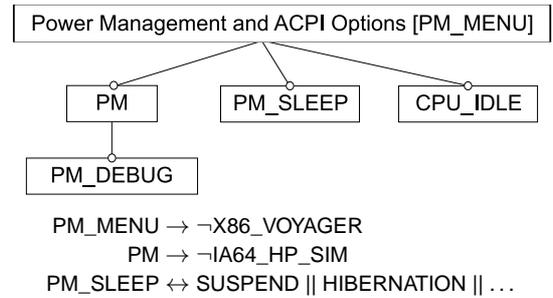


Fig. 3. Feature model for the example of Fig. 2

is m, the configs are constrained with OR. A choice marked as *optional* can be set to n and no choice config needs to be selected. Choices without the mark are *mandatory* and cannot be set to n. Tristate choices reflect a common binding variation: static binding requires an exclusive module to be linked statically; dynamic binding allows multiple alternative modules to be compiled—a single module is loaded at runtime.

Kconfig as a Feature Modeling Notation. We will interpret the hierarchy of configs, menus, and choices as the *Linux feature model*. Table I maps Kconfig concepts to feature modeling concepts. Figure 3 shows the feature model for the Kconfig example of Figure 2 generated by this mapping.

Switch configs map to optional features (Table I); each feature for a tristate config additionally has an attribute of type bmode, defined as enum {y,m}. An entry-field config maps to a mandatory feature with an attribute of an appropriate type, integer or string.

We map conditional menus to optional features; a more faithful translation would require extending the feature modeling notation with visibility conditions. Unconditional menus map to mandatory features.

We map a choice to a feature with a group underneath containing the grouped features representing the choice configs. A mandatory choice maps to a mandatory feature with an XOR-

TABLE I
MAPPING KCONFIG MODELS TO FEATURE MODELS

Kconfig concepts	Feature modeling concepts
Switch config	Optional feature
Entry-field config	Mandatory feature
Conditional menu	Optional feature
Unconditional menu	Mandatory feature
Choice	
Mandatory	⌞ Mandatory feat. + XOR-group
Optional	⌞ Optional feat. + XOR-group
Mandatory tristate	⌞ Mandatory feat. + OR-group
Optional tristate	⌞ Optional feat. + OR-group
Choice config	Grouped feature
Config, menu or choice nesting	⌞ Sub-feature relation
Visibility conditions	
Selects	⌞ Cross-tree constraint
Constraining defaults	

group. An optional choice maps to an optional feature with an XOR-group. In a slightly lossy manner, we map a tristate choice to a feature with the less restrictive OR-group.

III. THE LINUX VARIABILITY STUDY

We analyzed aspects of features, hierarchy, constraints, and natural-language content in the Linux model both quantitatively and qualitatively. When applicable, we applied the same analysis to a corpus of published models and compared the results with Linux.

Statistics gathering for Linux. Our statistics were gathered for the 2.6.28.6 version of the kernel. The Kconfig model was extracted from a normalized form using the kernel configurator code. Prior to gathering the statistics, we apply a post-processing stage to account for some special cases in the Kconfig model (e.g. there were 11 multiply-defined features in the hierarchy).

Corpus of published models. The corpus contains 32 models available from the SPLOT website [9] (see Appendix A). Most of these models are academic examples; from workshop and conference publications and MSc and PhD theses. They span many domains, including insurance, entertainment, web applications, home automation, search engines, and databases. Nineteen models represent software product lines; eight represent other types of product lines, e.g., hardware or business; and five represent entire domains, e.g., electronic commerce systems. Only five models describe real, existing software systems, however even these appears to be results of research efforts as opposed to regular industrial engineering practice.

To analyze the published models, we created a simple tool, reusing Mendonca’s parser for SPLOT’s XML-based feature model format (<http://www.splot-research.org/sxfrm.html>).

A. Characterization of Features

Feature and group statistics. Table II gives statistics for both Kconfig concepts and their feature modeling counterparts. The Linux model has 5426 features, which is an order of magnitude more than *Electronic Shopping*, the largest of the published models, with 287 features. The vast majority (4744 or 89%) are user-selectable. Only about 3% (188) of features have integer and String attributes. Note that 71

TABLE II
LINUX FEATURE AND GROUP STATISTICS

Kconfig Concept	Features	Mand.	Grouped	XOR + OR
Config	5323	0	146	0
Non / User-Sel.	547 + 4744			
Boolean	2005	0	136	0
Tristate	3130	0	10	0
Int	132	132	0	0
Hex	29	29	0	0
String	27	27	0	0
Menu	71	38	0	0
Choice	32	31	0	30 + 2
Total	5426	257	146	30 + 2

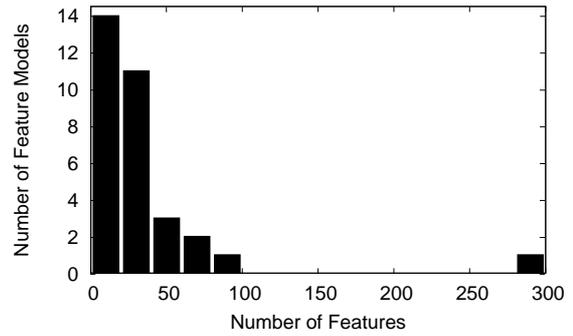


Fig. 4. Sizes of the published models

features correspond to menus, and 38 of these features are mandatory. Choices contributed 32 features and 32 groups; 30 of them are XOR-groups; the two OR-groups were contributed by two tristate choices.

Compared to published models (Figure 4, Table III) the Linux model contains very small percentages of mandatory features (5%), grouped features (3%), and groups (1%; relative to the number of features). Thus, the Linux features are mostly optional (92%).

Code-granularity of features. In order to assess the code-granularity of an average feature in Linux, we computed the set of source files included in the *allno* and in the *allyes* configurations. *Allno* tries to approximate the smallest possible configuration of Linux kernel, while *allyes* tries to approximate the largest configuration (both are included as targets in the build system and are based on a very simple algorithm). Table IV reports that a mere 61 user-selectable features are included in the former, and 3448 in the latter (which is about 73% of all user-selectable features). Moreover, all features except one of *allno* are also features of *allyes*. We used the difference between the sizes of the two configurations to compute an average feature size. An average feature spans 2.76

TABLE III
PUBLISHED MODELS VS. LINUX

% relative to no. features	Published Models median, min - max	Linux
mandatory features	25%, 0% - 66%	4.74%
grouped features	44%, 0% - 75%	2.69%
groups	16%, 0% - 35%	0.59%
	XOR	9%, 0% - 30%
	OR	6%, 0% - 16%
cross-tree constraint ratio	19%, 0% - 62%	82%

TABLE IV
CONFIGURATION STATISTICS

Metric	allno	allyes	Δ	θ
Features	61	3,448	3387	1
Files	973	10,326	9,353	2.76
SLOC	210,302	4,266,171	4,055,869	1,197.48

$$\Delta_i = \text{allyes}_i - \text{allno}_i; \theta_i = \Delta_i / \Delta_1$$

source files, and roughly a thousand non-blank non-comment lines of code (although surprisingly small, this number is still an over-approximation of the actual average, as it ignores the fact that lines not belonging to *allyes* are removed by the preprocessor).

Qualitative characteristics. To understand types of features and options in the Linux kernel, we performed a subjective categorization of 180 randomly selected features. The selected categories characterize the granularity of features from the user’s point of view—whether a feature enables support for a device or its option—as well as their type—whether the feature is related to a driver, protocol, API, etc.

We categorized features based on the help descriptions provided in the Kconfig files and by querying the web when needed. We left features with no description information in Kconfig uncategorized. As this type of classification is subjective, we performed a sanity check by cross checking the categories for 18 features. We found a discrepancy of only 8% and so believe that the categorization is sound and relevant. We used the following user-based granularity categories:

- *Menu:* grouping features, e.g., `IO_SCHEDULERS`, which groups read/write schedulers for block devices;
- *Support:* features that support certain devices or protocols, e.g., `HID_SAMSUNG`, which enables support for Samsung’s InfraRed remote control;
- *Option:* features enabling/disabling a specific kernel or driver capability, e.g., `DASD`, which enables DASD block devices using `S/390s` channel commands;
- *Debug:* features enabling tracing and other debug functions for developers, like `BOOT_TRACER`, which activates collection of run-time informations to assist boot time optimization; or `MEMSTICK_DEBUG`, which activates debug info for memory stick devices.

Additionally, we categorized features into the following types:

- *API:* features that provide an API for programming, e.g., `CRYPTO_CTR`, which enables API for the block cipher algorithm, required for IPsec;
- *Driver:* features related to drivers, e.g., `SND_ADLIB`, that enables support for AdLib cards;
- *Kernel:* change kernel behaviour, e.g., `FAILSLAB`, which enables fault-injection capability for `kmalloc`;

TABLE V
USER-BASED GRANULARITY CATEGORIES OF LINUX FEATURES

Menu	Support	Option	Debug	No Info
1	97	46	13	23

TABLE VI
TYPES OF FEATURES IN LINUX KERNEL

API	Driver	Kernel	Protocol	Subsystem	No Info
5	120	15	14	1	25

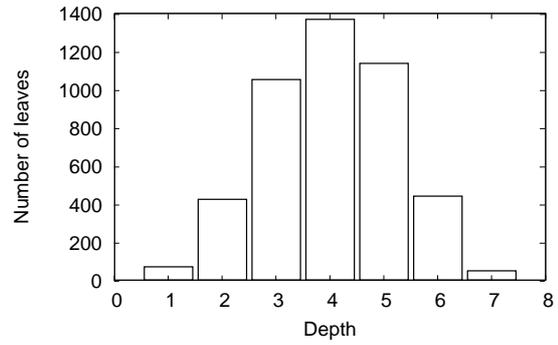


Fig. 5. Linux leaf depth

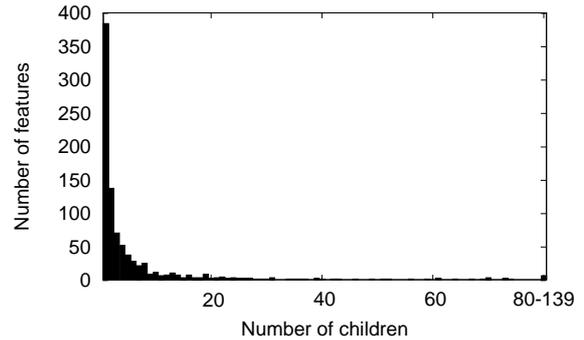


Fig. 6. Linux branching factor

- *Protocol:* features that implement protocols, e.g., `LLC2`, which enables support for `PF_LLC` sockets;
- *Subsystem:* features that enable whole subsystems, e.g., `Bluetooth`, `BT`, which enables the Bluetooth subsystem, comprising of several layers of software.

We found a very high correlation between support features and drivers, meaning that roughly 50% of features in the Linux kernel are drivers that support certain devices and protocols, greatly outnumbering features that enable smaller-grained functionality.

Interesting is also the relatively high number of the developer-oriented debug features.

B. Model Hierarchy

Figure 5 shows the number of leaves with a given depth for Linux. The maximum leaf depth in the Linux model is 7 (we assume level 0 for root). The maximum depth for published models is 10. The shapes of the leaf-depth histogram for the published models vary significantly; however, the shape for the largest published model, *Electronic Shopping*, has closest resemblance to that of Linux. An interesting observation is that the Linux model is relatively shallow (average depth of 4).

Figure 6 shows the number of features for a given *branching factor*, i.e., number of children, for Linux. The vast majority of features are leaves (4544; not shown in the histogram). Surprisingly, as many as 384 features are single-child parents;

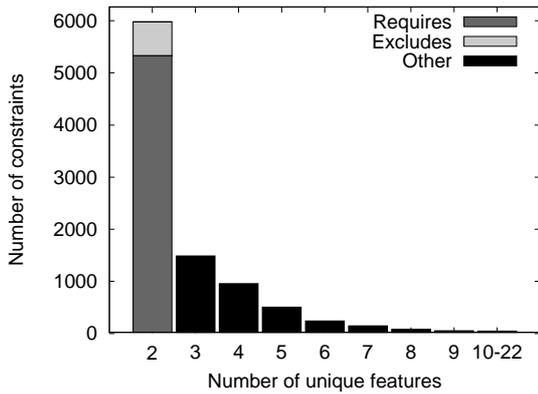


Fig. 7. Constraints categorized by the number of features they reference

these include features representing choices and singular options. The histogram has a long tail: 58 features have between 20 and 139 children. In contrast, the published models have maximum branching of 11, which is vastly smaller than for Linux. Interestingly, when generating models, White et al. [2] assume a branching factor of at most 5, which appears very low in the face of our data.

As before, histogram shapes of the published models vary significantly, with the shape for Electronic Shopping resembling most closely that of Linux. A notable difference, when compared to Linux, is the relatively low number of single-child parents in Electronic Shopping, where it is far lower than the number of two-child parents.

C. Constraints

A *constraint* restricts legal combinations of features. Obviously, hierarchical dependencies, discussed in the previous section are constraints. Not all dependencies can be expressed as hierarchical dependencies, however. Additional *cross-tree constraints*, specified in a suitable logic, are typically added besides the nesting structure. The Kconfig language includes a language of boolean expressions extended with equality predicates on non-boolean values for this purpose.

We determined three sources of cross-tree constraints in the Kconfig model: *visibility conditions*, *select clauses*, and *constraining defaults* (see Section II). The excerpt from Fig. 2 maps to the cross-tree constraints shown under the diagram in Fig. 3. The constraint on PM_DEBUG is not a cross-tree constraint—it belongs to the hierarchy. Also, the default in CPU_IDLE does not give rise to any cross-tree constraints, since it is not constraining itself, but merely proposes a default value that can be overridden.

As Mendonca [4] points out, the hardness of analysis of feature models lies in the complexity of their cross-tree constraints. At the same time, it is not uncommon that researchers believe that cross-tree constraints are rare and not crucial to feature modeling. For example in our reference corpus, eleven models do not have *any* cross tree constraints, and a further eleven only have one or two constraints. Only 37% of models in our sample have a significant amount of cross-tree

dependencies. In the Kconfig model, a total of 4186 (77%) of features declare a constraint in their definitions, sometimes more than one, giving rise to a total of 9291 constraints in the feature-modeling sense (top-level conjuncts).

It is interesting to see how these conjuncts distribute over more standard types of constraints (see Figure 7). 5313 (89%) of them are positive implications (also known as *requires* constraints), and 649 (11%) are *excludes* constraints (i.e. constraints of the form $f \rightarrow \neg g$). 3324 constraints represent more complex relations involving more than two features. The most complex of these includes 22 distinct literals in one constraint:

```
(MWINCHIP3D ∨ MCRUSOE ∨ MEFFICEON ∨ MCYRIXIII ∨
MK7 ∨ MK6 ∨ MPENTIUM4 ∨ MPENTIUMM ∨ MPENTIUMIII ∨
MPENTIUMII ∨ M686 ∨ M586MMX ∨ M586TSC ∨ MK8 ∨
MVIAC3_2 ∨ MVIAC7 ∨ MGEODEGX1 ∨ MGEODE_LXV
MCORE2) ∧ ¬X86_NUMAQ) ∨ X86_64) → X86_TSC = y
```

In contrast, published models almost exclusively contain binary constraints. Only 4 (12%) models in the sample contain a constraint relating 3 features, and none contained larger constraints. We should mention here that for analysis tools, binary constraints are easy. Consistency checking of a 2-CNF formula can be done in polynomial time, while consistency checking for a formula with ternary clauses is NP-complete. Thus the Linux model sets a new challenge for analysis tools.

Not only does the Linux kernel model contains a large number of constraints, but these constraints also involve unusually many features. Mendonca [4], [10] introduces *cross-tree constraints ratio* (CTCR)—a normalized measure comparing the number of features participating in cross-tree constraints (more precisely a percentage of features in the model that are referenced in constraints). As we can see in Table III, CTCR for published models varies between 0 and 60%, with 19% being a typical value. In the Linux model, all 82% of features participate in cross tree constraints. Effectively the Linux hierarchy says relatively little about the combinatorial dependencies between features, indicating a certain limitation of hierarchical models for describing dependencies in very complex software projects.

D. Natural Language Properties

Natural language processing techniques gain popularity in software engineering tools, including feature modeling [11], [12]. It is thus relevant to investigate the main properties of texts available in the Kconfig model. The Linux model has three kinds of textual attributes: feature identifiers (like PM), prompt texts (Power management support), and descriptions ("Power Management" means that ...).

Available Textual Information. We have counted the length of feature identifiers (considering strings separated by underscores as separate words), of prompts, and of descriptions. Table VII summarizes the findings. The majority of identifiers are 13 characters long, but there do exist some of length 2 (such as MD, SX, VT), and some of length 43

(SECURITY_SELINUX_POLICYDB_VERSION_MAX_VALUE). The majority of identifiers contains two or three words, with some approaching as many as nine. Most prompts include 3–5 words, with some reaching up to 13.

The help descriptions exhibit a similar pattern. The majority is around 30 words long, but some reach as much as 392 words. At the same time 823 features have no descriptions at all. Out of these, 540 are non-user-selectable, and 102 are menus or choices. As many as 181 of user-selectable configs contain no descriptions.

These numbers demonstrate a considerable effort of kernel developers to make features descriptions valuable for users. Still, about 20% of features have poor data like unhelpful identifiers or descriptions under 20 words. Most short descriptions are not very explanatory, containing texts such as *Say Y here* or *If unsure say N* or just the full name of the supported device. Longer descriptions have detailed explanations, such as when to enable the given feature and suggestions of other related features to enable or disable.

Vocabulary. We analyzed the most frequent domain specific terms in the Linux model. We consider any word not included in the *aspell* (0.60.5) English dictionary to be a domain term. Table VIII shows the most frequent domain words for the text attributes. Most common terms clearly relate to popular hardware kinds or to kernel subsystems.

Moreover, identifiers of 3601 features share common words with identifiers of their immediate parents (not necessarily domain specific words). For example: CRYPTO_DEV_HIFN_795X_RNG is a child of CRYPTO_DEV_HIFN_795X, or DVB_USB_DIBUSB_MB_FAULTY is a child of DVB_USB_DIBUSB_MB. Consequently, for about half of the features, it should be possible to automatically determine their immediate parents using a string similarity metric.

IV. THREATS TO VALIDITY

External Validity. Our baseline corpus is comprised of 32 models selected from published sources. Due to their academic origin, these models are not realistic. However, they are a suitable baseline here, as they support our main point that

TABLE VII
SIZE OF TEXTUAL ARTIFACTS IN THE KCONFIG MODEL

artifact	no. of characters			no. of words		
	median	min	max	median	min	max
identifiers	13	2	58	2	1	9
prompts	27	2	82	4	1	13
description	-	-	-	29	2	392

TABLE VIII
TOP DOMAIN TERMS IN THE KCONFIG MODEL

Text source	Most frequent domain terms					
Identifier	<i>usb</i>	<i>snd</i>	<i>md</i>	<i>serial</i>	<i>fb</i>	<i>debug</i>
Prompt	<i>usb</i>	<i>ethernet</i>	<i>pci</i>	<i>intel</i>	<i>scsi</i>	<i>pcmcia</i>
Description	<i>usb</i>	<i>linux</i>	<i>scsi</i>	<i>ethernet</i>	<i>pci</i>	<i>howto</i>

there exist realistic models that do not share characteristics with published models.

Studying the Linux kernel as a single subject raises a doubt whether the reported values of metrics are representative. However, we do not make any general claims about feature models based on these metrics. Rather we postulate that this model, which describes a realistic and wide-spread software system, should be included in benchmark sets of feature models. While a single model cannot be claimed to be representative, it *does* witness a departure from expected characteristics. Moreover, as Tartler [13] suggests, it is unlikely that Linux is an exception among operating systems, since other have similar qualities.

Arguably, the Kconfig model has not been created as a feature model but rather as a specification of a configuration system. Our interpretation of this model as a feature model can be seen as a violation of the original intention. Nevertheless Kconfig shares structural characteristics with feature models, and its main purpose (modeling a range of configuration choices) is consistent with the main objective of feature modeling. Unlike typical feature models, which are created during domain analysis, the Kconfig model has been grown bottom up—by adding features during evolution of the system. While these two processes, domain analysis and evolution—are different—the evolutionary bottom up construction is a realistic scenario, resembling the processes during evolution of mature product lines.

Internal Validity. The qualitative classification of Linux features has been done manually by one member of the project team for a randomly selected subset of 180 features. In order to ensure the representativeness of this result we have selected the 180 feature sample uniformly; in the sense that every feature in the kernel had equal probability to be selected. For a subsample of uniformly selected 18 features (out of the 180) we have independently verified the classification using another member of the project.

No formal definition of the Kconfig language is available, besides the only existing implementation itself. Thus, we could not verify whether our transformation of the Kconfig model to the feature modeling notation is semantics preserving. In order to decrease the chance of misinterpretations, we have used the original Linux configurator code to perform the first phase of that translation consisting of normalizing the representation and removing any syntactic sugar. Thus, we are confident that the first phase of the translation is in agreement with whatever the kernel developers have intended. For the remaining part of translation usual best practice measures were applied, such like investigating test cases using our translator against the Linux configurator tool.

Because there is no single accepted canonical syntactic representation for constraint systems, counting constraints is always subject to applied translations. In our case, however, it is very clear that regardless of which of the reasonable translations was used, the complexity of constraints in the Linux model is much higher than any constraints in the

published models.

V. RELATED WORK

The connection between the Kconfig language and feature modeling was previously described by Sincero and Schröder-Preikschat [14]. In their paper, they describe feature modeling concepts in terms of Kconfig constructs. Our work differs in that we investigate the inverse mapping—from Kconfig to feature model.

Tartler et al. [13] propose an architecture for a tool to detect dead features in the Linux Kernel. They make the same assumption as we do, i.e. that Kconfig can be naturally and meaningfully interpreted as a feature model. However, they are not interested in the properties of the model itself, except for existence of dead features. They extend a classic feature model analysis (dead feature detection) to the mapping of features to code to diagnose quality errors in the Linux codebase. It will be interesting to see if their analysis actually scales to the model we have described here.

Segura and Ruiz-Cortés [1] complain about the lack of realistic benchmarks for feature model analysis tools, and postulate creating a common standardized benchmark set with a repository of realistic models. Our contribution can be seen as one step towards addressing their concern, by providing a large and realistic feature model, which is now available online.

Randomly generated benchmarks have been applied in evaluation of analyzes tools for feature models [3], [2], [4], [10]. The generators applied in these projects have been tuned to generate models that resemble published models. However these characteristics turn out not to be true for large real life models, as exemplified by the Linux model described in this paper. Our results can serve as an input to design a statistically significant generator of random benchmarks for tools.

The European Project AMPLE has investigated use of automatic information retrieval techniques for extraction of feature models from requirement documents [11], [12]. The applicability and tuning of such algorithms to practical models heavily depends on properties of documentation available such as the size of provided descriptions and the amount of domain specific terms in them. The statistics of Section III-D are hopefully inspiring for this purpose.

Earlier we have presented reverse engineering algorithms that can be used to obtain examples of feature models from systems, where no explicit model is embedded—directly from compatibility constraints [15], or from sets of legal configurations [16].

VI. CONCLUSION

The Linux model is an excellent example of a large-scale variability model used in practice. We have shown that it challenges many of the long-held assumptions in the product-line community. It offers a wealth of information and acts as an excellent benchmark for the evaluation of automated analysis tools.

REFERENCES

- [1] S. Segura and A. Ruiz-Cortés, “Benchmarking on the automated analyses of feature models: A preliminary roadmap,” in *VaMoS*, 2009, pp. 9–17.
- [2] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. R. Cortés, “Automated diagnosis of product-line configuration errors in feature models,” in *SPLC*. IEEE Computer Society, 2008, pp. 225–234.
- [3] P. Trinidad, D. Benavides, A. R. Cortés, S. Segura, and A. Jimenez, “Fama framework,” in *SPLC*. IEEE Computer Society, 2008, p. 359.
- [4] M. Mendonca, A. Wąsowski, and K. Czarnecki, “Sat-based analysis of feature models is easy,” in *SPLC’09*. IEEE, 2009.
- [5] R. Zippel and numerous contributors, “kconfig-language.txt,” seen 2009-11-23.
- [6] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, “Feature-oriented domain analysis (FODA) feasibility study,” Technical Report CMU/SEI-90-TR-21, 1990.
- [7] K. Czarnecki and U. W. Eisenacker, *Generative Programming: Methods, Tools, and Applications*. Boston, MA: Addison-Wesley, 2000.
- [8] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk, “Is The Linux Kernel a Software Product Line?” in *Workshop SPLC-OSSPL 2007*, 2007.
- [9] M. Mendonca, M. Branco, and D. Cowan, “S.P.L.O.T.: software product lines online tools,” in *OOPSLA Companion*. ACM, 2009, <http://www.splot-research.org>.
- [10] M. Mendonca, A. Wasowski, K. Czarnecki, and D. D. Cowan, “Efficient compilation techniques for large scale feature models,” in *GPCE’08*, 2008, pp. 13–22.
- [11] N. Weston, R. Chitchyan, and A. Rashid, “A framework for constructing semantically composable feature models from natural language requirements,” in *SPLC*. IEEE Computer Society, 2009.
- [12] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler, “An exploratory study of information retrieval techniques in domain analysis,” in *SPLC*. IEEE Computer Society, 2008, pp. 67–76.
- [13] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, “Dead or alive: Finding zombie features in the linux kernel,” in *FOSD*, 2009, pp. 81–86.
- [14] J. Sincero and W. Schröder-Preikschat, “The linux kernel configurator as a feature modeling tool,” in *Proceedings of the Workshop on Analyses of Software Product Lines (ASPL)*, 2008, pp. 257–260.
- [15] K. Czarnecki and A. Wąsowski, “Feature models and logics: There and back again,” in *SPLC ’07*. IEEE, 2007.
- [16] K. Czarnecki, S. She, and A. Wąsowski, “Sample spaces and feature models: There and back again,” in *SPLC’08*. IEEE, 2008.
- [17] *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*. IEEE Computer Society, 2008.

APPENDIX A

CORPUS OF PUBLISHED MODELS

The corpus includes 32 models downloaded from the SPLIT web-site [9] on 16 November 2009:

Aircraft PL, Arcade Game PL, Car PL, Cellphone, CFDP Library, Connector PL, Digital Video System, Documentation_Generation, Electronic Shopping, FAME-DBMS, Graph, Graph Product Line, HIS, Insurance Policy, Insurance_Product, Inventory, James, JPlug, Key_Word_In_Context_index_systems, Model_Transformation, Monitor_Engine_System, MoviesApp PL, SAL, Search_Engine_PL, Smart Home, Stack PL, Telecommunication_System, Text_Editor, Thread, Virtual_Office_of_the_Future, Weather Station, Web_Portal

A Preliminary Review on the Application of Feature Diagrams in Practice

Arnaud Hubaux, Andreas Classen*
 PReCISE Research Centre
 University of Namur, Belgium
 Email: {ahu, acs}@info.fundp.ac.be

Marcílio Mendonça
 University of Waterloo, Canada
 Email: marcilio@csg.uwaterloo.ca

Patrick Heymans
 PReCISE Research Centre
 University of Namur, Belgium
 Email: phe@info.fundp.ac.be

Abstract—For two decades, feature diagrams have been intensively studied as a means to specify variability and pilot configuration in software product line engineering. Surprisingly though, it seems that very few reports on the use of feature diagrams in practice are available. To test this claim, we started a systematic review of such reports. In the collected material, we tried to identify positive and negative feedback on the use of feature diagrams. In this paper, we present the first results of this work in progress and discuss the opportunity of extending it to a fully systematic review on a wider scale.

I. INTRODUCTION

Software product line (SPL) engineering (SPLE) has long been promoted as a cost-effective means to build customisable products out of reusable assets [1]. SPL development is traditionally a two-activity process [2]. The first activity, called *domain engineering*, consists in developing a set of reusable assets that can be configured and combined to create different products of the SPL. A key point in domain engineering is the identification and documentation of variability of the SPL. Typically, the variability is documented in a variability model. Variability models can take different forms and shapes, and address artifacts of different natures [2]. Here we focus on a particular kind of model, namely *feature diagrams* (FDs) [3], which are typically used to provide a technology-independent and high-level representation of variability. The second activity is *application engineering* during which variability is progressively resolved: the stakeholders decide which features from the FD are selected for inclusion in the final product and which are discarded until the product is completely *configured* [4].

The research community has worked intensively on FDs for more than two decades. This resulted notably in regular enhancements of their expressiveness (e.g. [4]), formalisation (e.g. [5]) and automated reasoning tools (e.g. [6]). However, thorough studies of the fitness of FDs wrt. industrial problems are hard to find. By fitness we mean the ability to fulfil a particular function or meet a particular need. Our discussions with researchers and practitioners also support this “informal observation”. In this context, we believe it is necessary to clear up the matter and provide evidence of the presence or absence of supportive material. This concern can be translated into the following research question: *What evidence do we have of the fitness of FDs in practice?*

*FNRS Research Fellow.

In order to answer the research question, one possibility is to conduct a systematic literature review [7]. However, the time and resources needed to perform such a review are considerable. It is then often recommended to start with a smaller scale review that serves as an opportunity and feasibility study for a full systematic review [8]. In this paper, we follow this recommendation and propose a preliminary review based on a limited sample of paper sources.

Another systematic review in the field of variability modelling was published recently by Chen *et al.* [9] but our objectives are different:

- we focus on FDs and not on variability models in general;
- we inventorise *applications* of FDs in practice rather than approaches to model variability.

The main contribution of this paper is preliminary evidence that industrial application of FDs has been barely tackled in the literature and that there is an opportunity to conduct a full systematic review. In the long run, the expected benefits from our study are the following:

- a comprehensive inventory of success stories as well as failures of FDs in industry that can be used to define guidelines to help practitioners decide whether FDs are appropriate or not to their specific needs;
- repeated updates of our study will allow to assess the progress of the acceptance of FDs in industry;
- the collected observations can be used to identify the major research problems and define a practice-driven research roadmap for FD modelling and analysis.

The paper is structured as follows. Section II presents the review method and Section III describes the results of our analysis. Section IV discusses the limitations of our results and opportunities for conducting a full systematic review.

II. REVIEW METHOD

Since this study is only meant to be a pilot for a possible full systematic review, we relaxed some of the guidelines proposed by Kitchenham [7]. We thus refer to this pilot as a semi-systematic review. We will elaborate more on this in Section IV.

The initial paper base consists of the proceedings of all editions of (1) the *software product line conference* (SPLC) and co-located workshops for the years 2000, 2002, and

2004–2009, (2) the workshop on *product family engineering* (PFE) for the years 2001 and 2003, and (3) the workshop on *Variability Modelling of Software-intensive Systems* (VaMoS) for the years 2007–2009. PFE has been retained to ensure the continuity of publications between 2000 and 2009, as suggested on the SPLC history web page.¹ We focus on these three events because they are major events for the SPL community and already total 415 papers.² Therefore, we think that this sample is representative of the activity in the field, which is confirmed by [9].

From those 415 papers, a first short list of 29 papers³ was established based on their titles. This short list was obtained after a filtering based on a disjunction of three search criteria. The first criterion aims at capturing papers that study FDs and their applications, e.g. product configuration, in real-life examples. In order to avoid papers that only consider toy examples, we included a set of terms that usually refer to real cases. They are detailed below. The second and third criteria broaden the scope by considering papers that discuss the application of SPLs in practice but which title do not contain an explicit reference to a topic different from variability modelling (e.g. architecture modelling). Note that given the nature of these latter two criteria, the search of papers was done manually. To be accepted, the title of the paper had to match at least one of the criteria, i.e. contain:

- **(Criterion 1)**
at least one of the terms *feature diagram*, *feature model*, *variability model*, *configuration or derivation*
and
the name of a company or a reference to a practical application (e.g. *industry*, *practice*, *case study*, *empirical or experience*) or a reference to an economic context (e.g. *market or finance*);
- **or (Criterion 2)**
one of the terms *product line* or *product family*
and
the name of a company or a reference to a practical application or a reference to an economic context
and
no reference to a topic different from variability modelling;
- **or (Criterion 3)**
the name of a company or an industrial application domain
and
no reference to a topic different from variability modelling;

The 29 papers that matched the criteria are listed in Table I and sorted by criterion. Note that one paper ([11]) matched more than one criteria.

¹<http://splc.net/history.html>

²Apart for SPLC'09 and the three editions of VaMoS for which we directly investigated the proceedings, the lists of published papers was retrieved from DBLP on the 20th of October 2009.

³Unfortunately, the access to paper [10] was not granted.

TABLE I
DISTRIBUTION OF THE SELECTED PAPERS BY CRITERIA.

Criterion	Number of papers	References
1	9	[10], [11], [12], [13], [14], [15], [16], [17], [18]
2	20	[19], [20], [11], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37]
3	1	[38]

The set of 29 papers was then further filtered based on their abstracts and introductions. Only those whose abstract or introduction addressed the evaluation or application of variability models in industrial settings, or discussed the applicability of FD-based configuration systems were kept. At the end of the filtering process, 16 papers remained (bold-faced in Table I), i.e., roughly 4% of the total number (415) of papers.

At this point we did not know what results to expect when reviewing the papers. Therefore, data extraction was performed in an *ad hoc* way (by collecting notes of possibly relevant information), instead of using a data extraction form that we should have defined *a priori*, as suggested by Kitchenham [7]. Similarly, we defined the paper categories, which are discussed in the next section, after having read the papers.

III. FINDINGS

A. Classification of the papers

Our readings lead us to identify four categories. The first one relates to successful applications of FDs for which some adaptations to the FD language were made. The second one denotes successful applications for which no adaptations to the language were made. The third one gathers cases where the authors do not actually use FDs but acknowledge that they could have helped. The fourth one discusses unsuccessful applications. A last category was added to gather false positives, i.e., papers for which the applications of FDs turned out to be missing or too vague to tell anything about their fitness. The distribution of the papers we reviewed is summarised in Table II.

TABLE II
DISTRIBUTION OF THE REVIEWED PAPERS BY CATEGORIES.

Category	Number of papers	References
Successful applications <i>with</i> adaptations	2	[17], [16]
Successful applications <i>without</i> adaptations	4	[37], [33], [31], [14]
FDs could have helped	3	[11], [15], [13]
Unsuccessful applications	2	[22], [12]
False positives	5	[35], [34], [21], [19], [18]

B. Description of the papers

We now describe the content of all the papers belonging to each of the categories defined above.

Successful applications of FDs with adaptations. Gillan *et al.* [17] report on their application of an experimental FD notation applied to the modelling of embedded software in the telecommunication domain. The three main challenges to the adoption of FDs were (1) the management of the growing number of variation points, (2) the need to implement features formerly available in the software into the hardware, and (3) the specification of feature behaviours. For each of these challenges, they propose extensions to FDs suited to the telecommunication domain. The design of their system architecture is based on this extended version of FD language.

Reiser *et al.* [16] aim at specifying a unified FD language based on FODA [3]. According to their experience in the automotive industry, they claim that the biggest threats to using FDs in complex SPLs are the heterogeneity of development techniques and the divergence of subsystems. The framework they propose to address these threats uses hierarchically organised FDs linked to the other artifacts. In addition, they present a list of seven requirements for a unified language collected from their experience in the automotive industry. They also touch upon two potential benefits of FDs in this context: (1) features can link management, marketing and development within and between companies, and (2) FDs provide a central view of variability over a vast range of artifacts.

Successful applications of FDs without adaptations. Dordowsky *et al.* [37] discuss the adoption of SPL principles to manage the software variants and technology variations in complex avionic systems. One of their achievements is the efficient production of source code for different variants based on a FD. Unfortunately, very little detail about the FD and the way code is generated from it are available. They also report that there is a pressing need for higher integration between the FD and the other constituents of the tool chain.

Jensen [33] looks into the derivation of products in the domain of intelligence planning, collection and analysis for the US government and its allies. The author reports that FODA [3] helped to define the domain analysis process but gives no indication how FODA actually contributed to the final design. The author also complains about the lack of tool support to handle variation points across the tool chain.

Steger *et al.* [31] report on the introduction of SPLE at Bosch Gasoline Systems. They used FDs to model the variability of the SPL but do not provide feedback about the actual advantages of FDs in their application, though they mention the lack of available tool support. By applying feature analysis their goal was to reduce resource consumption, a critical aspect in embedded systems.

Wenzel *et al.* [14] explain how FDs can be used to tailor the databases of the configuration management system available in the IBM Tivoli suite. In their case, FDs proved to simplify and ease the understanding of the stakeholders who lacked specific knowledge about the underlying structure of the databases.

Furthermore, using FDs for configuration reduced the “*error-prone elicitation of requirements*” and enabled the automation of choice propagation. A quick evaluation of their approach revealed that (1) the tree-based navigation coupled with cross-cutting constraints “*were regarded as a significant advantage*”, (2) FD-based database specification has a “*time-saving potential*” and (3) the reduced amount of knowledge needed to understand the database can potentially increase customer acceptance. About this latter point though, they mention that experts might miss information intentionally left out of the FD.

Applications where FDs could have helped. Deelstra *et al.* [11] studied two large industrial organisations and identified a collection of product derivation problems, including the lack of hierarchical structuring of variation points, and the lack of formal representation of variation points and dependencies. The authors clearly refer to FDs as a potential solution to the former but do not elaborate further on how to solve the other problems.

O’Leary *et al.* [15] compare two approaches for product derivation in industrial settings. A result of their comparison is a list of lessons learned for product derivation. A strong emphasis is put on the need to represent variability differently according to the type of stakeholder who is dealing with it. The authors do not rule out FDs as a potential representation but they add that FDs should not be the only one available.

Schmid *et al.* [13] studied three existing configuration tools unrelated to SPLE and compared them according to the variability concepts usually present in FDs. Although their investigation does not allow to draw any conclusion regarding the application of FDs, they provide evidence that the FD and configuration tool worlds overlap and “*co-evolved in a similar way*”.

Unsuccessful applications of FDs. Ishida [22] discusses the application of SPLE at the Nomura Research Institute to develop semi made-to-order software packages. The author rejects FD-based approaches as they “*may produce more problems than benefits*” because of the “*degree of software intensiveness [sic], the ambiguity of criteria to decompose systems into features, and the frequency of requirement specification changes*”. The author adds that the key to success is abstraction rather than massive configuration of concrete artefacts. Their approach follows the model-view-controller (MVC) decomposition, and is based on a combination of UML, entity-relationship diagram and the Turbine web application framework.

Tolvanen *et al.* [12] identify approaches to define domain-specific modelling (DSM) languages that enable automated product derivation in practice. According to them, FDs are clearly not a good candidate as they “*operate at a level too general to identify DSM concepts*” to be included in the language and “*do not capture the dependencies and constraints required to define modelling constructs*”. Instead, they recommend to work at the level of the architectural model, which better supports the identification of product concepts and their relationships.

False positives. Carbon *et al.* [35] focus on the integration of SPLE principles in existing workflows and infrastructures to facilitate the customization of office devices without referring to any variability model in particular. Habli *et al.* [34] elaborate mainly on the role and definition of an appropriate configuration management plan to develop products. Although relevant to the field, they do not connect them to FDs. Helferich *et al.* [21] discuss the distinction between marketed and engineered SPLs but do not discuss the impact of this distinction on FDs. Jaring *et al.* [19] identify several variability issues taken from their experience with various industrial partners and advocate a reference framework normalising the representation of variability throughout the SPL development lifecycle. They do not explicitly point to FDs as a concrete solution. Wnuk *et al.* [18] focus on the management of variability at the requirements level. Aside from a brief reference to OVMs [2], which are compared to their “product configuration specification”, they do not discuss the use of variability models.

C. General observations

We start with the observations that directly address our research question:

- **few papers on the application of FDs were found.** We first observe that less than 2% of all the papers (8 out of 415) actually discuss successful and unsuccessful applications of FDs in practice. Among these 8 papers, we observe 6 successful and 2 unsuccessful reports. This very small number of applications makes it difficult to draw any conclusion, expect that it tends to confirm our impression that there are very few experience reports. The low number of unsuccessful reports is, however, more understandable as practitioners are generally reluctant to publish unsuccessful attempts.
- **few details about the usage of FDs were found.** Out of the 6 *successful* reports, only 3 ([17], [16], [14]) provide details about how FDs were used. Unfortunately, the advantages and disadvantages described are still preliminary and the gains of using FDs are not backed up by concrete evidence. The papers that suggest FDs as a *potential* solution do not really substantiate this choice either. The first paper ([22]) reporting on *unsuccessful* uses of FDs also fails to provide substantial evidence. Its observation seems to be based on speculation rather than on facts, its justification being that “[FDs] may produce more problems than benefits” [22]. The second case of the *unsuccessful* applications ([12]) affirms that FDs were not suited but again does not detail the evaluation process that lead to this conclusion.
- **lack of existing material is corroborated by other sources.** For instance, a systematic review of variability management (VM) approaches published at SPLC 2009 [9] concludes:

There is only little, if any, experimental or detailed comparative analysis to show the relative

advantages and disadvantages of different VM approaches. That is why it would be hard to build an evidence-based guidance for selecting a VM approach for specific development situation and context. Hence, there is a vital need of conducting comparative analysis of different approaches in order to provide the practitioners with a qualified portfolio of techniques.

FDs being part of VM approaches, their survey also provides the insight that comparative evaluations of FDs’ advantages and disadvantages are lacking, too. Deelstra *et al.* [11] and Jaring *et al.* [19] go along the same lines saying that most of the approaches meant to support product derivation “fail to provide substantial supportive evidence” [11]. Gillan *et al.* [17] declare that the acceptance of FDs in telecommunication requires “more and deeper case studies”.

Other interesting observations we made are that there is a:

- **growing interest from practitioners.** What Table II does not show is that 8 of the 11 papers reporting on applications of FDs have been published between 2007 and 2009, i.e. more than two thirds of the experiences were reported during the last two years. Even though this trend still has to be confirmed by more comprehensive studies, it seems to show a growing interest for FDs in practice.
- **lack of tool support.** 3 papers ([31], [33], [37]) complain about the lack of tool support and the weak integration among the constituents of the tool chain. Although this information does not really help to understand how FD languages should be improved, it stresses the urge to develop dependable and easily interoperable tools. If not the key to the uptake of FDs, it would at least facilitate their evaluation.
- **lack of relationships among modelling languages.** The results of Schmid *et al.* [13] provide a first feeling that the configuration tool world and SPLE have somehow evolved in parallel but with converging goals. However, they also observed that a stronger emphasis was put on the relationship between the artifacts (e.g. components and abstract features) in configuration systems than in SPLE. As for tool support, this pinpoints a trend in SPLE research to focus on a modelling language and develop model-centred reasoning without much care for interoperability. Providing more integration among languages would be an opportunity to increase the acceptance of FDs by practitioners.

Concerning the different dialects of FDs used in the successful cases, there is unfortunately not much to say either. Gillan *et al.* [17] and Steger *et al.* [31] use their own dialect and developed specific tool support. Reiser *et al.* [16] used pure::variants and seem to use the FD language proposed by Czarnecki [4] in their examples. Jensen [33] vaguely references FODA whereas Dordowsky *et al.* [37] do not mention any particular FD dialect. Wenzel *et al.* [14] used the feature

modelling plug-in for Eclipse from Antkiewicz *et al.* [39]. This multitude of dialects does not allow to draw any conclusion regarding a preference for a particular language. Furthermore, the lack of justification makes it hard to understand the rationales for their selection. The only noticeable fact is that in four of the six cases, tools were either explicitly developed ([17], [31]) or used ([16], [14]).

Before concluding, we mention two additional observations that both address the relevance of our search criteria.

- **false positives after abstract and introduction.** Table I shows that, out of the 9 papers matching the first criterion, 8 were selected for review, i.e., we had only one false positive. The second criterion is, however, far less discriminant as only 9 of the 20 (i.e. half of the papers) were selected for a complete review. This is probably due to the broader scope of this latter criterion. The third criterion turned out to be inconclusive as the only paper matching this criterion was not selected for a full review.
- **false positives after full review.** The filtering based on the abstract and introduction still resulted in 5 false positives, i.e., only 11 papers turned out to be relevant to our research question, as shown in Table II. Here, it is interesting to note that, among these 5 papers, 4 matched the second criterion whereas only 1 matched the first. This confirms the previous observation that the second criterion is too general.

The outcome of our review provides a fairly disappointing answer to the research question. It shows that for three of the most important venues in the field, the available material is sufficient neither to convince of the relevance of FDs nor to let practitioners evaluate whether FDs are a suitable solution to their problems. It also reveals that the current lack of tool support and interoperable languages might be major barriers to the acceptance of FDs in industry. Yet, the recent growing interest of practitioners should encourage researchers to actively publish their experience report to constitute a strong body of knowledge.

IV. DISCUSSION

We now discuss the threats to the validity of our preliminary review, and the opportunity and feasibility of conducting a full systematic review.

A. Threats to validity

In Section II, we referred to our review process as *semi-systematic*. According to Kitchenham [7], a *systematic* review should follow a two-step process. The first step is the *planning* of the review. During planning, the objective and the review protocol are defined. In our case, the objective was intentionally modest as our study is meant to be a pilot, and hence more of an exploratory nature than an exhaustive collection of evidence. Consequently, our review protocol is a simplified version of the one suggested by Kitchenham, i.e. no quality assessment was performed, the data extraction protocol was not systematic and no detailed metadata analysis was carried out. Also, the protocol was not reviewed by external experts.

It is during the second step that the review is actually *conducted*. In our lightweight review process, we considered a limited set of paper sources, that were reviewed by only one researcher (the first author), without any external expert being involved in this task. In addition, all the paper venues were considered to be of equal importance during our analyses. Finally, no predefined data collection forms were used to record the results.

The impact of these limitations on the quality of our analyses is hard to tell given the small sample space we considered. However, the goal of this review was to study the feasibility of a full systematic review. The conclusions we drew previously should therefore be seen as preliminary, too.

B. Opportunity and feasibility of a full systematic review

The findings reported in the previous section point to a lack of experience reports for FDs in three important publication venues. We also observed that, in the experience reports, the justifications for the claimed advantages and drawbacks of FDs are often quite thin. Our preliminary findings might lead to the conclusion that academic research on FDs is out of touch with reality in software engineering. Several possible reasons for this lack of experience reports are conceivable. First, FDs are used but practitioners barely report on their applications. Possible causes are that practitioners do not want to publish their FDs, do not want to pay their engineers to write papers discussing their experience with FDs, or do not have sufficient incentive for publishing their experiences. Secondly, practitioners do not want to advertise unsuccessful applications of FDs. Finally, FDs might simply not be used by practitioners.

Confirming this finding is important. This justifies the need for a thorough and systematic review of a broader scope. We envision several ways to broaden the scope of our research. Obviously, we could consider more software engineering venues and journals, including those not specifically dedicated to SPLE, like ICSE, RE, ASE, TOSEM or TSE. Similarly, we could broaden the scope to include industrial venues. A completely different path would be to explore other engineering domains that also have to model variability and deal with configuration issues.

In order to increase our chances of collecting valuable results, we intend to specify a full-fledged systematic review protocol. Besides alleviating the threats to validity discussed above, we have to learn the lessons from our observations. We mentioned in our review method that the paper classification was not defined *a priori* but *a posteriori*. We now have to look at the classification we defined from a distance and evaluate how its refinement could facilitate the analysis of the results. Section III-C clearly showed that the second criterion was too general and lead to many false positive papers. Note that it was only applied to events dedicated SPLE and, thus, had a limited relevance. Further investigations are still needed to tell whether a broader scope of research will justify the refinement of this criterion. In contrast, the third criterion will probably be too general for papers outside the SPL community and will have to be refined.

The review of papers from other domains is likely to call for a completely different review protocol since its goal would be to identify well accepted techniques similar to FDs (rather than identify usages of FDs in other domains). Therefore, another pilot study will probably be required to first identify the techniques used that are comparable to FDs. Based on these results, another systematic review could identify those that are used in practice in their respective domains.

In case a systematic review confirms the preliminary results observed here, it should probably be followed by an investigation of the reasons for the absence of experience reports (e.g. conduct a large-scale survey of practice in industry).

V. CONCLUSION

In this paper, we questioned the availability of evidence supporting the fitness of feature diagrams in practice. In order to answer this question, we conducted a semi-systematic literature review. This review focused on two of the major venues for software product line research, i.e. the software product line conference (SPLC) and the workshop on variability modelling of software-intensive systems (VaMoS). Our preliminary findings demonstrate a lack of solid evaluation of the impact of feature diagrams on the industry. The review, however, is still preliminary, semi-systematic and limited in scope. Its results, although negative, are encouraging and call for a more thorough and systematic review, which is planned future work. In the meantime, we hope to encourage more empirical research on FDs and urge researchers to publish existing empirical results.

ACKNOWLEDGEMENTS

This work is sponsored by the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy, under the MoVES project, and the FNRS.

REFERENCES

- [1] P. C. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, ser. SEI Series in Software Engineering. Addison-Wesley, August 2001.
- [2] K. Pohl, G. Bockle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, July 2005.
- [3] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," SEI, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, November 1990.
- [4] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Formalizing cardinality-based feature models and their specialization." *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.
- [5] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Generic semantics of feature diagrams," *Computer Networks*, p. 38, 2006.
- [6] M. Mendonça, "Efficient reasoning techniques for large scale feature models," Ph.D. dissertation, University of Waterloo, 2009.
- [7] B. A. Kitchenham, "Procedures for undertaking systematic reviews," Computer Science Department, Keele University (TR/SE-0401) and National ICT Australia Ltd (0400011T.1), Tech. Rep., 2004.
- [8] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," *Journal of Systems and Software*, vol. 80, no. 4, pp. 571 – 583, 2007.
- [9] L. Chen, M. Ali Babar, and N. Ali, "Variability management in software product lines: A systematic review," in *SPLC'09*, San Francisco, CA, USA, 2009, pp. 81–90.
- [10] A. Hein, M. Schlick, and R. Vinga-Martins, "Applying feature models in industrial settings," in *SPLC'00*. Norwell, MA, USA: Kluwer Academic Publishers, 2000, pp. 47–70.
- [11] S. Deelstra, M. Sinnema, and J. Bosch, "Experiences in software product families: Problems and issues during product derivation," in *SPLC'04*, Boston, MA, USA, 2004, pp. 165–182.
- [12] J.-P. Tolvanen and S. Kelly, "Defining domain-specific modeling languages to automate product derivation: Collected experiences," in *SPLC'05*, Rennes, France, 2005, pp. 198–209.
- [13] K. Schmid and C. Kroher, "An analysis of existing software configuration systems," in *DSPL'09*, San Francisco, CA, USA, 2009, pp. 2–7.
- [14] S. Wenzel, T. Berger, and T. Riechert, "How to configure a configuration management system - an approach based on feature modeling," in *MAPLE'09*, San Francisco, CA, USA, 2009, pp. 99–105.
- [15] P. O'Leary, R. Rabiser, I. Richardson, and S. Thiel, "Important issues and key activities in product derivation: Experiences from two independent research projects," in *SPLC'09*, San Francisco, CA, USA, 2009, pp. 121–130.
- [16] M.-O. Reiser and M. Tavakoli, R. and Weber, "Unified feature modeling as a basis for managing complex system families," in *VaMoS'07*, Limerick, Ireland, 2007, pp. 79–86.
- [17] C. Gillan, P. Kilpatrick, I. Spence, R. Gawley, J. Brown, and R. Bashroush, "Challenges in the application of feature modelling in fixed line telecommunications," in *VaMoS'07*, Limerick, Ireland, 2007, pp. 141–148.
- [18] K. Wnuk, B. Regnell, J. Andersson, and S. Nygren, "An industrial case study on large-scale variability management for product configuration in the mobile handset domain," in *VaMoS'09*, Seville, Spain, 2009, pp. 155–164.
- [19] M. Jarig and J. Bosch, "Representing variability in software product lines: A case study," in *SPLC'02*, London, UK, 2002, pp. 15–36.
- [20] M. Raatikainen, T. Soinen, I. Männistö, and A. Mattila, "A case study of two configurable software product families," in *PFE'03*, 2003, pp. 403–421.
- [21] A. Helfferich, K. Schmid, and G. Herzurm, "Reconciling marketed and engineered software product lines," in *SPLC'06*, 2006, pp. 23–27.
- [22] Y. Ishida, "Software product lines approach in enterprise system development," in *SPLC'07*, Kyoto, Japan, 2007, pp. 44–53.
- [23] D. Sellier and G. Benguria, G. and Urchegui, "Introducing software product line engineering for metal processing lines in a small to medium enterprise," in *SPLC'07*, Kyoto, Japan, 2007, pp. 54–62.
- [24] Y. Matsumoto, "A guide for management and financial controls of product lines," in *SPLC'07*, Kyoto, Japan, 2007, pp. 163–170.
- [25] R. Kreuter, C. Lescher, A. Schreiber, and C. Schwanninger, "Applying a cost model for product lines: Experience report," in *MESPUL'08*, 2008, pp. 263–271.
- [26] M. Khurum, T. Gorschek, and K. Pettersson, "Systematic review of solutions proposed for product line economics," in *MESPUL'08*, 2008, pp. 277–284.
- [27] A. J. Nolan, "Building a comprehensive software product line cost model," in *SPLC'09*, 2009, pp. 249–256.
- [28] Y. Takebe, N. Fukaya, M. Chikahisa, T. Hanawa, and O. Shirai, "Experiences with software product line engineering in product development oriented organization," in *SPLC'09*, 2009, pp. 275–283.
- [29] R. Buhrdorf, D. Churchett, and C. W. Krueger, "Salion's experience with a reactive software product line approach," in *PFE'03*, 2003, pp. 317–322.
- [30] J. Snyder, H. Lai, S. Reddy, and J. Wan, "Software product line support in coremetrics oa2004," in *SPLC'04*, 2004, pp. 188–191.
- [31] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber, "Introducing pla at bosch gasoline systems: Experiences and practices," in *SPLC'04*, Boston, MA, USA, 2004, pp. 34–50.
- [32] R. Kolb, I. John, J. Knodel, D. Muthig, U. Haury, and G. Meier, "Experiences with product line development of embedded systems at testo ag," in *SPLC'06*, 2006, pp. 172–181.
- [33] P. Jensen, "Experiences with product line development of multi-discipline analysis software at overwatch textron systems," in *SPLC'07*, Kyoto, Japan, 2007, pp. 35–43.
- [34] I. Habli and T. Kelly, "Challenges of establishing a software product line for an aerospace engine monitoring system," in *SPLC'07*, Kyoto, Japan, 2007, pp. 193–202.
- [35] R. Carbon, S. Adam, and T. Uchida, "Towards a product line approach for office devices - facilitating customization of office devices at ricoh co," in *SPLC'09*, San Francisco, CA, USA, 2009, pp. 151–160.

- [36] W. J. Slegers, "Building automotive product lines around managed interfaces," in *SPLC'09*, 2009, pp. 257–264.
- [37] F. Dordowsky and W. Hipp, "Adopting software product line principles to manage software variants in a complex avionics system," in *SPLC'09*, San Francisco, CA, USA, 2009, pp. 265–274.
- [38] F. r. van de Linden and J. G. Wijnstra, "Platform engineering for the medical domain," in *PFE'01*, 2001, pp. 224–237.
- [39] M. Antkiewicz and K. Czarnecki, "Featureplugin: feature modeling plug-in for eclipse," in *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*. New York, NY, USA: ACM, 2004, pp. 67–72.

Cardinality-Based Feature Modeling and Model-Driven Engineering: Fitting them Together

Abel Gómez¹, Isidro Ramos²

Department of Information Systems and Computation
Universidad Politécnica de Valencia
Valencia, Spain

¹agomez@dsic.upv.es

²iramos@dsic.upv.es

Abstract—Feature Modeling is a technique which uses a specific visual notation to characterize the variability of product lines by means of diagrams. In this sense, the arrival of metamodeling frameworks in the Model-Driven Engineering field has provided the necessary background to exploit these diagrams (called feature models) in complex software development processes. However, these frameworks (such as the Eclipse Modeling Framework) have some limitations when they must deal with software artifacts at several abstraction layers. This paper presents a prototype that allows the developers to define cardinality-based feature models with constraints. These models are automatically translated to Domain Variability Models (DVM) by means of model-to-model transformations. Thus, such models can be instantiated, and each different instantiation is a configuration of the feature model. This approach allows us to take advantage of existing generative programming tools, query languages and validation formalisms; and, what is more, DVMs can play a key role in MDE processes as they can be used as inputs in complex model transformations.

Keywords-Software Product Lines; Model Driven Architecture; Feature Modeling; UML; OCL

I. INTRODUCTION

The key aspect of Software Product Lines (SPL) [6] that characterizes this approach against other software reuse techniques is how to describe and manage variability. Although several approaches have addressed this problem, the most of them are based on feature modeling, proposed in [12]. In this approach, the commonalities and variabilities among the products of a SPL are expressed by means of the so-called features (*user-visible aspect or characteristic of the domain*), which are hierarchically organized in feature models.

The use of feature models can be exploited by means of metamodeling standards. In this context, the Model-Driven Architecture [14] proposed by the Object Management Group is a widely used standard which arises as a suitable framework. In this sense, the Meta Object Facility (MOF) and the Query/Views/Transformations (QVT) standards allows us to define feature models and their instances, and use them in a Model-Driven Engineering (MDE) process. In this context, MDE and the Generative Programming approach [7] provides a suitable basis to support the development of SPLs. Moreover,

Generative Programming and SPLs facilitate the development of software products for different platforms and their use under different technologies.

In this paper we discuss the main issues that arise when trying to use feature models in a MDE process, and how to easily overcome them. We also present a tool that allows the developers of SPLs to define, use and exploit feature models in a modeling and metamodeling tool. In our case, we have chosen the Eclipse Modeling Framework (EMF). Moreover, the EMF framework provides several tools which permit us to enrich these models (by means of OCL expressions and OCL interpreters) and to deal easily with them (by using model transformations). All these features of EMF allows us to use this framework to start a Software Product Line.

The remainder of this paper is structured as follows: in section II we present the starting point of our work and the main problems that arise when trying to use feature models in a MDA process; and in sections III and IV we present both the ideal approach and the practical approaches to use feature modeling in a MDA context. In section V we show our feature metamodel and how we operationalize the solution presented in section IV in our prototype tool. Related works are discussed in section VI and in section VII we present our conclusions and future works.

II. FOUNDATIONS

A. Cardinality-based feature models at a glance

Cardinality-based feature modeling [8] integrates several of the different extensions that have been proposed to the original FODA notation [12]. In this sense, a cardinality based feature model is also a hierarchy of features, but the main difference with the original FODA proposal is that each feature has associated a *feature cardinality* which specifies how many clones of the feature are allowed in a specific configuration. Cloning features is useful in order to define multiple copies of a part of the system that can be differently configured.

Moreover, features can be organized in *feature groups*, which also have a *group cardinality*. This cardinality restricts the minimum and the maximum number of group members that

can be selected. Finally, an *attribute type* can be specified for a given feature. Thus, a primitive value for this feature can be defined during configuration.

In feature models is also quite common to describe constraints between features such as the *implies* and the *excludes* relationships, which are the most common used ones. According to the original FODA notation, these constraints can be expressed by means of propositional formulas [2], thus, it is possible to reason about the satisfiability of the feature model. As explained in [8], this interpretation for feature models is not very adequate when dealing with cardinality-based feature models due to the fact that we can have multiple copies of the same feature. Therefore, it is necessary to clearly define the semantics of the constraint relationships in the new context, where features can have multiple copies, and features can have an attribute type and a value. In this case, we need more expressive approaches to (i) define constraints between features and (ii) perform formal reasoning over the feature models and their constraints.

B. Feature models and their configuration

A configuration of a feature model is usually defined as *the set of features that are selected from a feature model without violating any of the constraints defined in it*, but it can also be defined as *a valid set of instances of a feature model*. I.e., the relationship between a feature model and a configuration is comparable to the relationship between a class and an object. The first definition, is well-suited when dealing with “traditional” feature models (those that can be defined by using the original FODA notation). In this case, every instantiation of the elements of the feature model will follow the *singleton pattern*, that is, every feature can have at most one instance. Fig. 1 shows an example of this.

In Fig. 1a an example feature model is represented. This feature model represents a system S , with two features A and B . The first one, feature A , is mandatory (it must be included in every possible product of the product line), and the second one, feature B , is optional (it can be included in a particular product or not). Thus, we have two possible configurations for this feature model, which are represented in figures 1b and 1c.

As can be seen, this process of selection of features (according to the defined constraints) is closely related with a copy mechanism, that is, a configuration of a feature model is a more restrictive copy of the original one that represents exactly one variant. This mechanism can also be used in cardinality-based feature models. In this case, when features have an upper bound greater than 1 they can be cloned at

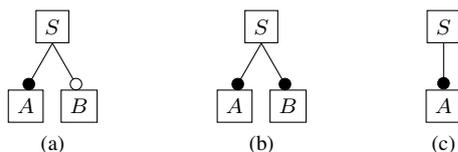


Fig. 1. Example of a feature model (1a) and the two possible configurations that it represents (1b and 1c).

model level, thus, we can have multiple *copies* of the same feature. Several specializations can be done at model level until only one variant is possible.

C. Cardinality-based feature models and MDE main issues

In previous paragraphs we have described what cardinality based feature models are and how they are usually exploited. Nevertheless, there are some issues, that we will address in the remaining of this paper, that cause some problems when trying to use feature models in a MDE process.

The first one comes from the classical definition of configuration of a feature model (the set of features that are selected from a feature model). This definition tends to define the configuration as a *copy* mechanism instead of as an *instantiation* mechanism. Although this definition can be somewhat intuitive when dealing with traditional feature models, it can be confusing when dealing with cardinality-based ones. In this case, the instantiation concept is best suited when talking about cardinality-based feature models with attribute types, given that the configuration is more easily understandable as an *instance-of* relationship rather than as a *copy-and-refinement-of* relationship.

The second issue that comes up is how to deal with model constraints when features can be cloned in our models and such features can have an attribute type. In this case, as pointed out in section II, we need to use more expressive languages that allows us (i) to deal with sets of features (i.e. n copies of feature A) and (ii) to deal with typed variables which values can be unbounded in order to easily represent attribute types.

III. PUTTING MODEL-DRIVEN ENGINEERING AND FEATURE MODELING TOGETHER

The Meta Object Facility standard (MOF, [16]), which provides support for meta-modeling, defines a strict classification of software artifacts in a four-layer architecture (from M3 to M0 layer). The meta-metamodel layer (M3) is where the MOF language is found. MOF can be seen as a subset of the UML class diagram, and can be used to define metamodels at M2 layer. This way, artifacts that reside in layer x , are expressed in terms of the constructors defined in layer $x + 1$.

In turn, the Query/Views/Transformations (QVT, [15]) standard describes how to provide support to queries and model transformations in a MDE process. QVT uses the pre-existent *Object Constraint Language* (OCL, [17]) language to perform queries over software artifacts. The QVT standard provides three different languages to describe these transformations. In this sense, the QVT-Relations can be interesting because of its implicit support for traceability and its high level of abstraction, as it is a declarative language. This language uses object templates to define relationships among different domains that can be enforced, performing a model transformation when needed.

As the MOF standard provides support for modeling and metamodeling, we can use it to define cardinality-based feature models by defining its metamodel. Previously, a configuration

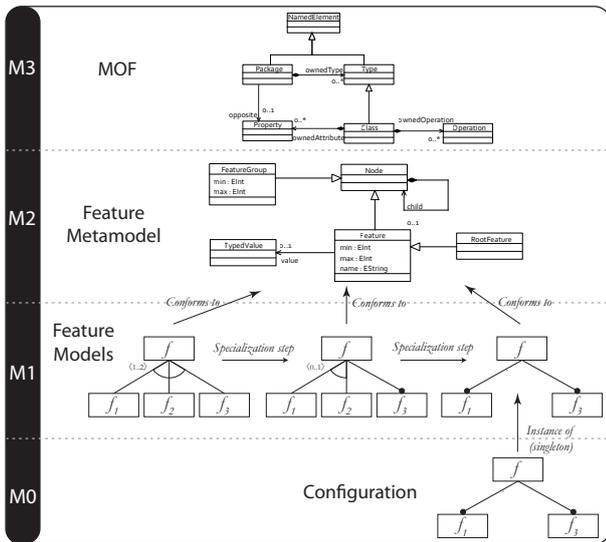


Fig. 2. Specialization and configuration of feature models in the context of MOF

process by means of specialization was shown. This conception about the configuration process involves copy of features, but it has a big implication: configurations are expressed in terms of the feature metamodel instead of in terms of the feature model. Fig. 2 shows how the specialization process fits in the four-layer architecture of MOF. In this figure the EMOF language is represented in a simplified way in the level M3. In the level M2 the metamodel for cardinality-based feature models is represented by using the MOF language (also in a simplified way), and finally, in level M1 some feature models are represented. The leftmost model is the one that represents our family of systems, and starting from it, we obtain the final model as an specialization of the original one. Thus, the configuration of the feature model is defined in terms of the features metamodel as the model it conforms to has no variability, and both feature model and configuration are practically equivalent.

In order to use a feature model in a MDE process, we need to use the one that captures the whole variability of the domain. This is necessary to define any possible configuration of the model, but it is also necessary in order to define model-based transformations that allows us to use this feature models and their configurations in other complex processes.

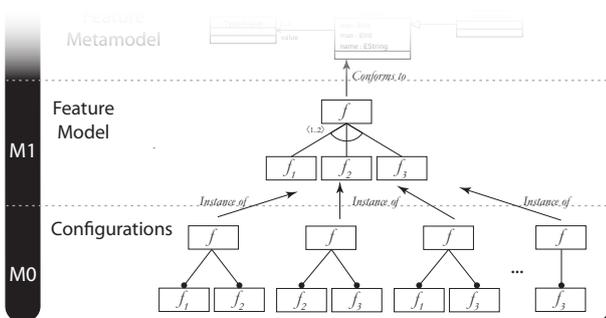


Fig. 3. Instantiation of feature models

Fig. 3 shows how a configuration without performing a specialization process looks like. This way, the feature model can be used in any modeling framework to automatically generate configuration editors, and what is more, feature models and configurations can take part of a MDE process. Developers can take advantage of related tools, feature models can be used to guide model transformations with multiple inputs, and configurations can be automatically checked against their corresponding models using built-in query languages.

IV. USING A MODELING FRAMEWORK FOR FEATURE MODELS AND THEIR CONFIGURATIONS

The Eclipse Modeling Framework (EMF) [10] can be considered as an implementations of the MOF architecture. Ecore, its metamodeling language can be placed at layer M3 in the four-layer architecture of the MOF standard. By means of Ecore, developers can define their own ecore models which will be placed at the metamodel layer (M2). An example of such metamodels can be the metamodel to build cardinality-based feature models. Finally, this Ecore models can be used to automatically generate graphical editors which are capable of building *instance models*, which will be placed at M1 layer. In the case of feature modeling, these *instance models* are the feature models. The left column on Fig. 4 shows this architecture.

As can also be seen in Fig. 4, the M0 layer is empty. That is a limitation of most of the modeling frameworks which are available today. As said, EMF provides a modeling language (Ecore) that can be used to define new models and their instances. This approach only covers two layers of the MOF architecture: the metamodel and the model layers. However, in the case of feature modeling we need to work with three layers of the MOF architecture: metamodel (cardinality-based feature metamodel), model (cardinality-based feature models), and instances (configurations).

Fig. 4 shows how to overcome this drawback: it is possible to define a model-to-model transformation in order to convert a feature model (i.e. the model represented by Feature model which can not be instantiated) to an Ecore model (i.e. the Domain Variability Model, DVM, which represents the Feature model as a new class diagram). Thus, it is possible to represent a feature model at the metamodeling layer, making

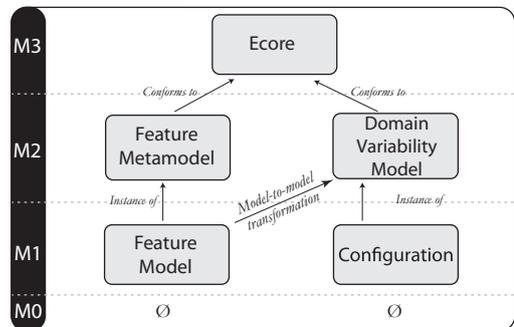


Fig. 4. EMF and the four-layer architecture of MOF

TABLE I
CARDINALITY-BASED FEATURE METAMODEL: PROPOSED TYPES OF
RELATIONSHIPS BETWEEN FEATURES

		Vertical (hierarchical) relationships	Horizontal relationships
Binary relationships	Mandatory		Biconditional 
	Optional		Implication 
Grouped relationships	Generic	$0 \leq j \leq k \leq m$ 	Exclusion 
	XOR	$0 \leq j \leq 1$ 	Use 
	OR	$0 \leq j \leq 1 < k \leq m$ 	

*where n is the number of childs

the definition of its instances possible. This way, developers can take advantage of EMF again, and automatically generate editors to define feature model configurations, and validate them against their corresponding feature models thanks to their new representation, the DVM.

V. OUR APPROACH

Based on the concepts presented in the previous section and using EMF, we have developed a tool that allows us to automate several steps in order to prepare a feature model that can be exploited to develop a SPL in the context of MDA. In this sense, our tool provides:

- Graphical support to define (a variant of) cardinality-based feature models.
- Automatic support to generate Domain Variability Models from feature models that capture all the variability of the application domain, allowing the developers to use them in model transformations.
- Automatic support for configuration editors, which will assist the developers in the task of defining new configurations.
- Capabilities to check the consistency of a configuration against its corresponding feature model.

A. Cardinality-based feature metamodel

The basis of our work is the cardinality-based feature metamodel, which permits to define feature models. In our proposal we have decided to represent explicitly the relationships between features. Thus, our metamodel represents in an uniform way the hierarchical relationships and the restrictions between features. Table I classifies and summarizes the types of relationships that the feature metamodel is able to represent. As can be seen, relationships are classified in two orthogonal groups:

- *Vertical vs. horizontal relationships.* Vertical relationships define the hierarchical structure of a feature model and

horizontal relationships define dependencies and restrictions between features.

- *Binary vs grouped relationships.* Binary relationships define relationships between two single features. In turn, grouped relationships are a set of relationships between a single feature and a group of childs.

Given this classification, the following relationships exist:

- *Binary and vertical relationships.* These relationships define structural relationships between two single features. In our approach, they represent a *has_a* relationship between a parent and a child feature. They can be mandatory and optional depending on the lower bound value. The upper bound (n) can be on both cases 1 or greater than 1, and indicates how many instances of the child feature will be allowed.
- *Grouped and vertical relationships.* Grouped and vertical relationships are a set of binary relationships where the child features share a *is_a* connotation with respect to their parent feature. A group can have an upper and a lower bound. These bounds specify the minimum and the maximum number of features that can be instantiated (regardless of the total number of instances).
- *Binary and horizontal relationships.* These relationships are specified between two features and do not express any hierarchical information. They can express constraints (biconditional, implications and exclusion) or dependencies (use). The first group applies to the whole set of instances of the involved features, however, the second one allows us to define dependencies at instance level, i.e.:

- *Implication* ($A \rightarrow B$): If an instance of feature A exists, at least an instance of feature B must exist too.
- *Coimplication* ($A \leftrightarrow B$): If an instance of feature A exists, at least an instance of feature B must exist too and vice versa.
- *Exclusion* ($A \times \text{---} \times B$): If an instance of feature A exists, can not exist any instance of feature B and vice versa.
- *Use* ($A \text{---} \rightarrow B$): This relationship will be defined at configuration level, and it will specify that an specific instance of feature A will be related to one (or more) specific instances of feature B as defined by its upper bound (n).

Fig. 5 shows our feature metamodel. Such metamodel has been defined taking into account that every element will have a different graphical representation. This way, it is possible to automatically generate the graphical editor to draw feature models based on such metamodel. In that figure, a feature model is represented by means of the FeatureModel class, and a feature model can be seen as a set of Features and the set of Relationships among them. A feature model must also have a root feature, which is denoted by means of the rootFeature role.

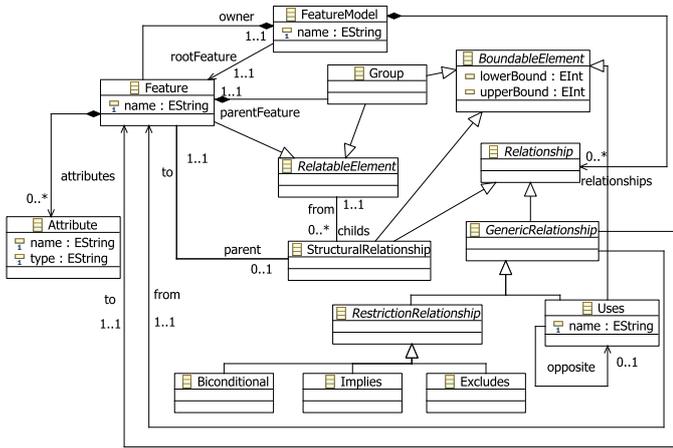


Fig. 5. Cardinality-based features metamodel

Binary relationships in table I are represented in the features metamodel as descendants of the Relationship class. Class StructuralRelationship represents the so called *Vertical* relationships and GenericRelationship represents the *Horizontal* ones. StructuralRelationships relate one parent RelatableElement (a Feature or a Group) with one child Feature. A Group specifies that a set of StructuralRelationships should be considered as a group.

It is noteworthy to point out two slight differences of our approach with respect to the classical cardinality-based feature models. First, we represent feature multiplicities at relationship level instead of at feature level (by means of the BoundableElement class). This allows us to easily define mandatory and optional relationships explicitly. Second, features can not have an attribute type. In turn, this information is expressed in terms of feature attributes. Feature attributes express information which is complementary to a feature and can be used to describe *parametric features*.

B. Cardinality-based feature modeling editor

The cardinality-based feature modeling editor allows us to easily define new feature models. Following the Model-Driven Software Development (MDS) approach, it has been automatically generated from the metamodel presented in the previous section. To obtain this graphical editor, the Graphical Modeling Framework (GMF [9]) has been used.

Fig. 6 shows what this editor looks like. The palette is located on the right side of the figure, and shows the tools that can be used to define the feature models. In the canvas an example feature model is shown. This feature model describes a product line for mobile phones. A mobile phone must have a screen, which can be touchscreen or not. Touchscreens can use resistive or capacitive technology. Moreover, a mobile phone can also support handwriting recognition, however, this feature is incompatible with normal screens. This can be described by means of an excludes relationship (solid line with crosses at its ends between feature Normal and HandwritingRecognition). In our product line, mobile phones can also have one or two cameras, each one with their corresponding resolutions. Those cameras can be used for videoconferencing or digital photography. If the camera is used for digital photography, it can be associated to an optional flash light. This is specified by means of the dashed line between features Camera and Flash. Finally, these mobile phones can have FM radio support, but it can be installed only if the device has a headphones connection, as the FM antenna is part of this accessory. This dependency is expressed by the implies relationship (directed line between feature Radio and feature Headphones).

C. Generating the Domain Variability Model

In section IV was explained that it is necessary to execute a model-to-model transformation in order to easily define configurations of a feature model in EMF. Following the MDA approach, this transformation has been defined by using the Relations language defined in the QVT standard. In order to integrate and execute the transformation process in our

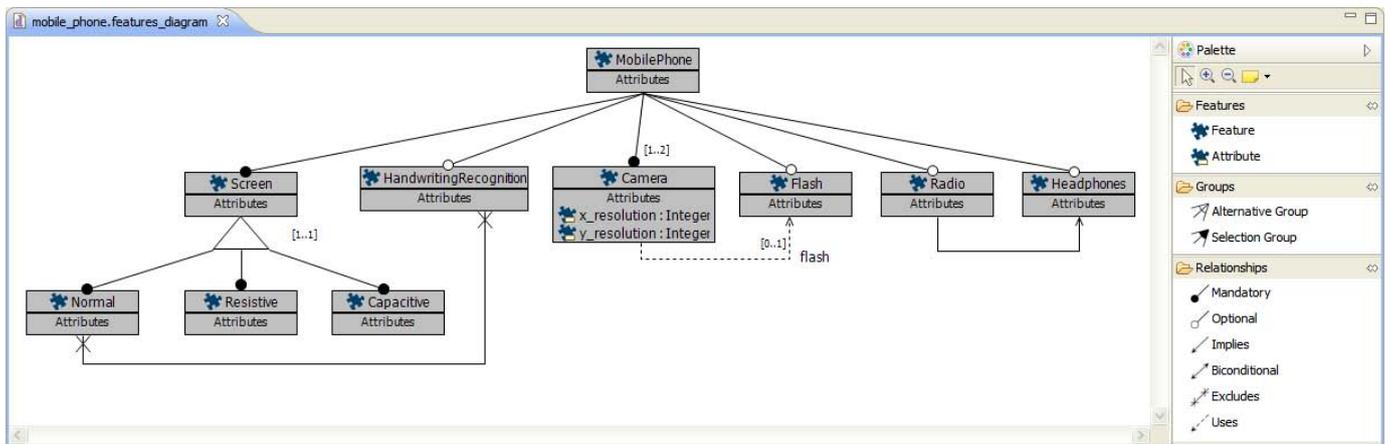


Fig. 6. Screenshot of the cardinality-based feature modeling editor

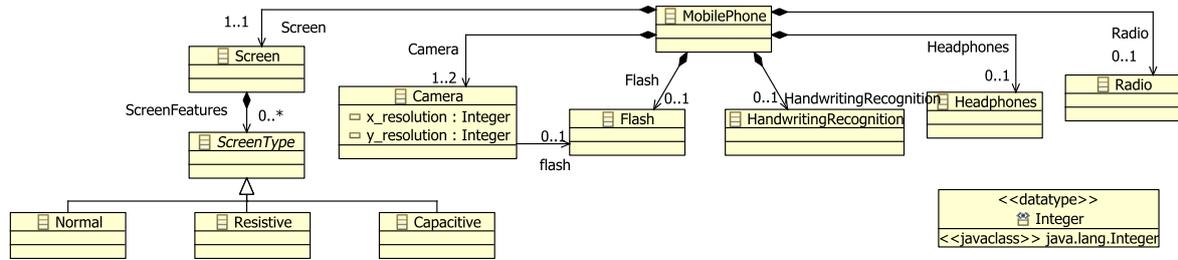


Fig. 7. Example Domain Variability Model

prototype, a custom tool based on the mediniQVT [11] transformations engine has been built.

The following paragraphs describe the transformation which transforms a feature model (expressed as an instance of the cardinality-based features metamodel in EMF), to a Domain Variability Model (DVM, expressed as an Ecore model that can be instantiated). The transformation is declared as follows and it is executed from the *feature* domain to the *classdiagram* domain.

```
transformation Feature2ClassDiagram(feature :
  features, classdiagram : ecore) { ... }
```

As feature models describe not only the structure of the features but also the relationships among them, it is necessary to define rules to generate both the structure of the DVM and the restrictions that apply to it. First, the structure of the DVM is defined by means of Ecore containment references and inheritance relationships; and second, restrictions are defined by means of OCL expressions. These OCL expressions are included on the DVM itself by means of *EAnnotations*. This *EAnnotations* are automatically used in next steps by our prototype to check that configurations are valid.

Figure 7 shows the resulting Ecore model. The rules that have been applied are:

- *Feature2Class*. For each Feature of the source model an *EClass* with the same name of the feature will be created. All these classes will be created inside the same *EPackage*, whose name and identifier derives from the feature model name. All the features in Fig. 7 are example of the application of this rule.
- *FeatureAttribute2ClassAttribute*. For each feature Attribute, an *EAttribute* will be created in the target model. This *EAttribute* will be contained in its corresponding *EClass*. Any needed *EDataType* will be also created. Attributes in the Camera *EClass* are example of this.
- *StructuralRelationship2Reference*. For each StructuralRelationship from a parent Feature a *containment* *EReference* will be created from the corresponding *EClass* (i.e. same name than the Feature). The multiplicity of this *EReference* will be the lower and upper bounds of the StructuralRelationship. *Containment* *EReferences* from MobilePhone are example of the application of the rule.
- *Group2Reference*. This rule states that for each Group contained in a Feature a *containment* *EReference* will be

created from the corresponding *EClass* (i.e., same name than the Feature). This *EReference* will point to a new abstract class, whose name will be composed by the Feature name and the suffix “Type”. This rule has the following post-conditions: *GroupChild2Class*, *Group2ChildsAnnot*, *GroupChild2LowerAnnot* and *GroupChild2UpperAnnot*. Screen, ScreenType and the *EReference* ScreenFeatures are example of the result produced by this rule.

- *GroupChild2Class*. This rule is in charge of creating the *EClasses* from the Features belonging to a Group. Moreover, each one of these *EClasses* inherit from the abstract *EClass* that has been previously created. *EClasses* Normal, Resistive, Capacitive, and their inheritance relationship are example of the application of the rule.
- *Group2ChildsAnnot*, *GroupChild2LowerAnnot*, and *GroupChild2UpperAnnot*. These rules create *EAnnotations* that will contain OCL expressions. This expressions will check that the multiplicities specified for the Group and the child Features are also satisfied by the instances of the DVM.
- *UsesRelationship2Reference*. For each Uses relationship between two Features, an *EReference* will be created in the target model. This *EReference* will relate two *EClasses* whose names will match the Features names. The *EReference* flash, between Camera and Flash, shows and example of this.

The next relations generate OCL expressions in the DVM for each restriction relationship of the source model. They can be easily expressed as the following OCL invariants created in the root *EPackage*:

- *ExcludesRelationship2ModelConstraint*. This rule generates the following invariant for (A excludes B):

```
A.allInstances()->notEmpty() implies B.
  allInstances()->isEmpty() and
  B.allInstances()->notEmpty() implies A.
  allInstances()->isEmpty()
```

- *ImpliesRelationship2ModelConstraint*. If the relationship is (A implies B), the following OCL expression is created by this rule:

```
A.allInstances()->notEmpty() implies B.
  allInstances()->notEmpty()
```

- *BiconditionalRelationship2ModelConstraint*. This rule creates the following OCL invariant if the source relationship is (A if and only if B):

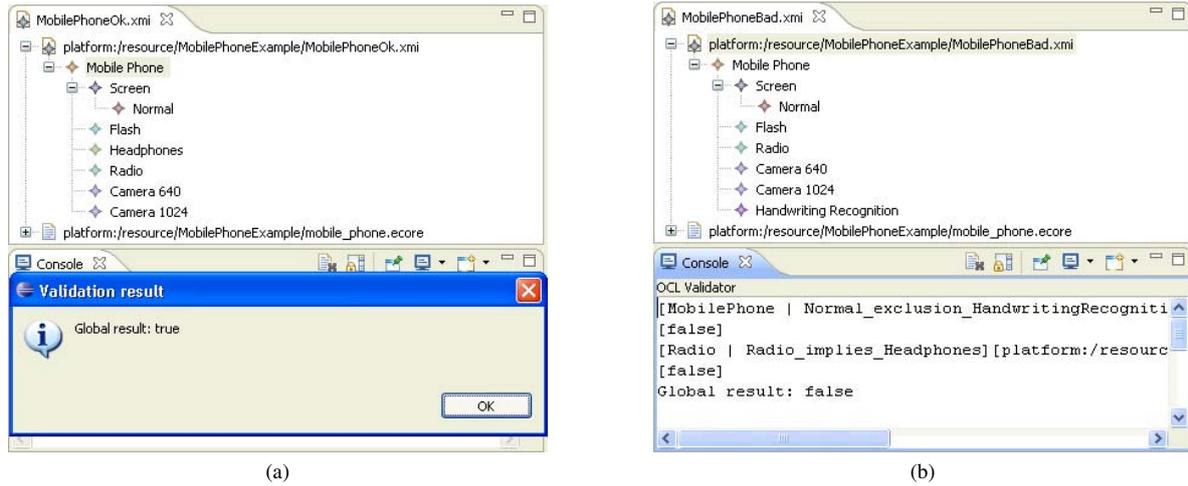


Fig. 8. Example of successful (8a) and unsuccessful (8b) configuration check

```
A.allInstances()->notEmpty() implies B.
allInstances()->notEmpty() and
B.allInstances()->notEmpty() implies A.
allInstances()->notEmpty()
```

D. Creating and validating configurations

In order to create new configurations of feature models it is not necessary to use any custom tool. As far as we have a DVM which captures the same variability than the original feature model, developers can use the standard Ecore tools. The most straightforward method to create a new configuration is to use the “Create Dynamic Instance...” option of the standard “Sample Ecore Model Editor”. This way, the “Sample Reflective Ecore Model Editor” can be used to define new configurations of our feature model. However, although EMF provides all the libraries and technologies to exploit OCL expressions, there is not a default method to check OCL invariants which are directly stored as EAnnotations in Ecore models themselves. Thus, we have built an extension which can take advantage of the OCL invariants that have been automatically created in the previous transformation step.

Fig. 8a shows an example configuration. A mobile phone with a normal screen, radio, headphones and two cameras has been defined. This configuration is valid conforming to the example feature model, as the popup window in the figure shows. However, Fig. 8b shows a configuration with a normal screen and handwriting recognition, which violates the excludes relationship. Moreover, this configuration includes radio support, but it does not include the headphones connection which is also invalid. When the configuration is invalid the checking process is unsuccessful. In this situation, the prototype console shows a summary with the constraints that are not met, and which are the problematic elements.

VI. RELATED WORKS

Feature modeling has been an important discussion topic in the SPL community, and a great amount of proposals for variability management have arisen. Specially, most of

them are based in the original FODA notation and propose several extensions to it [5]. Our work is closely related with previous research in feature modeling, however, there are several distinctive aspects:

In [8] a notation for cardinality-based feature modeling is proposed. In this sense, our tool shares most of this notation as it is widely known and used, but we have included some variants. First, in our approach features can not have an *attribute type*, but rather, they can have typed *feature attributes* which can be used to describe *parameterized features*. Second, in [8] both feature groups and grouped features can have cardinalities. However, the possible values for grouped features cardinalities are restricted. In our proposal, these values are not restricted and have different meanings: cardinality of feature groups specify the number of features that can be instantiated, and cardinality of grouped features specify the number of instances that each feature can have.

Our work describes a prototype to define configurations of feature models. Previous work has been also done in this area, such as the *Feature Modeling Plugin* [1]. This tool allows the user to define and refine a feature model and configurations by means of specializations. The advantage of this approach is that it is possible to guide the configuration process by means of constraint propagation techniques. The main difference with our work is that configurations are defined in terms of the feature metamodel and both models and configurations coexist at the same layer. Thus, in order to be able to deal both with models and configurations it is necessary to build complex editors (as they must guarantee that the specialization process is properly done).

Some previous works have already represented feature models as class diagrams. In [8] the translation from feature models to class models is performed manually, and no set of transformation rules are described. In this work, OCL is also presented as a suitable approach to define model constraints, but as the correspondences between feature models and class diagrams are not precisely defined, there is no automatic

generation of OCL invariants. In turn, [13] do present a set of QVT rules to automatically generate class diagrams from feature models. However, in this case, neither model constraints nor configuration definitions support is presented.

In [2] a proposal for feature constraints definition and checking is done. Specifically, this work proposes to represent features as propositions and restrictions among them as propositional formulas. However, in propositional formulas only `true` and `false` values are allowed. This approach is not suitable to our work, as we can have typed attributes which can not be expressed by this kind of formulas. Thus, we state that more expressive languages are needed. In this case, we propose OCL as our constraint definition language. Nevertheless, in order to perform more advanced reasoning (for example, satisfiability of feature models) richer formalisms are needed.

VII. CONCLUSIONS

In this paper we have proposed a prototype to define and use feature models in a MDE process. This prototype addresses one of the main issues that arises when dealing with nowadays metamodeling tools: they usually are not able to deal simultaneously with artifacts located in all the MOF layers. For example, the selected modeling framework (EMF) is only able to represent three different MOF layers (from M3 to M1), so that, our prototype defines a mechanism to overcome this typical limitation by means of model transformations. This way, feature models can be transformed to Domain Variability Models that can be instantiated and reused in future steps of the MDE process.

Our tool has been also designed following the MDE principles and a metamodel for cardinality-based feature modeling has been defined. Thus, by means of generative programming techniques, a graphical editor for feature models has been built. Feature models defined with this editor are automatically transformed in DVMs that are used to define configurations of feature models. Although several tools to define feature models and configurations in the last years have arised, our approach has several advantages against previous approaches: (i) the infrastructure that we propose to build configurations is simpler and more maintainable, as it is built following the MDSD guides; (ii) configurations are actually instances of a feature model (expressed by means of the DVM), so we can take advantage of the standard EMF tools; (iii) as feature models are described by DVMs that can be instantiated, both models and configurations can be used in other MDE tasks; (iv) having a clear separation between feature models and configuration eases the validation tasks as they can be performed by means of built-in languages; and (v) as the transformation between feature models and DVMs is performed automatically by means of a declarative language we can trace errors back from DVMs to feature models.

It is important to remark that having both feature models and configurations at different layers is very useful as they can easily be used in model transformations. In [3] an example is shown. In this work, a model transformation with multiple inputs is used to generate a software architecture automatically.

In this case, one of the arguments of this transformation is a configuration of a feature model, which is used to guide the architecture generation process.

Finally, to use DVMs allows us to address some satisfiability problems from new points of view. The introduction of cardinalities and unbounded attribute types makes harder to reason about feature models. Thus, richer formalisms (compared with the traditional ones) are needed. Fortunately, class diagrams are widely used and known, and several formalisms to reason about them have been proposed. In this sense, we have already done some preliminary works in model consistency checking by using formal tools [4], and future works are oriented towards this direction. A first version of our prototype can be downloaded from <http://issii.dsic.upv.es/~agomez/feature-modeling>.

ACKNOWLEDGMENTS

This work has been supported by the Spanish Government under the National Program for Research, Development and Innovation MULTIPLE TIN2009-13838 and the FPU fellowship program, ref. AP2006-00690.

REFERENCES

- [1] M. Antkiewicz and K. Czarnecki. FeaturePlugin: feature modeling plugin for Eclipse. *2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72, 2004.
- [2] D. Batory. Feature models, grammars, and propositional formulas. pages 7–20. Springer, 2005.
- [3] M. E. Cabello, I. Ramos, A. Gómez, and R. Limón. Baseline-oriented modeling: An mda approach based on software product lines for the expert systems development. *Intelligent Information and Database Systems, Asian Conference on*, 0:208–213, 2009.
- [4] J. Cabot, R. Clarisó, and D. Riera. Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 547–548, NY, USA, 2007. ACM.
- [5] L. Chen, M. A. Babar, and N. Ali. Variability management in software product lines: A systematic review. In *Proceedings of the 13th International Software Product Lines Conference (SPLC'09)*, San Francisco, CA, USA, 2009.
- [6] P. Clements, L. Northrop, and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, August 2001.
- [7] K. Czarnecki and U. W. Eisenacker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
- [8] K. Czarnecki and C. H. Kim. Cardinality-based feature modeling and constraints: A progress report, October 2005.
- [9] Eclipse Organization. The Graphical Modeling Framework, 2006. <http://www.eclipse.org/gmf/>.
- [10] EMF. <http://download.eclipse.org/tools/emf/scripts/home.php>.
- [11] ikv++ technologies AG. ikv++ mediniQVT website. <http://projects.ikv.de/qvt>.
- [12] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990.
- [13] M. A. Laguna, B. González-Baixauli, and J. M. Marqués Corral. Feature patterns and multi-paradigm variability models. Technical Report 2008/01, Grupo GIRO, Departamento de Informática, may 2008.
- [14] Object Management Group. MDA Guide Version 1.0.1. 2003. <http://www.omg.org/docs/omg/03-06-01.pdf>.
- [15] Object Management Group. MOF 2.0 QVT final adopted specification (ptc/05-11-01). 2005. <http://www.omg.org/cgi-bin/doc?ptc/2005-11-01>.
- [16] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification (ptc/06-01-01), 2006. <http://www.omg.org/cgi-bin/doc?formal/2006-01-01>.
- [17] Object Management Group. OCL 2.0 Specification. 2006. <http://www.omg.org/cgi-bin/doc?formal/2006-05-01>.

Integrating Automated Product Derivation and Individual User Interface Design

Andreas Pleuss, Goetz Botterweck, Deepak Dhungana
 Lero – The Irish Software Engineering Research Centre
 University of Limerick, Limerick, Ireland
 {andreas.pleuss, goetz.botterweck, deepak.dhungana}@lero.ie,

Abstract—Software Product Lines, in conjunction with model-driven product derivation, are successful examples for extensive automation and reuse in software development. However, often each single product requires an individual, tailored user interface of its own to achieve the desired usability. Moreover, in some cases (e.g., online shops, games) it is even mandatory that each product has an individual, unique user interface of its own. Usually, this results in manual user interface design independent from the model-driven product derivation. Consequently, each product configuration has to be mapped manually to a corresponding user interface which can become a tedious and error-prone task for large and complex product lines. This paper addresses this problem by integrating concepts from SPL product derivation and Model-based User Interface Development. This facilitates both (1) a systematic and semi-automated creation of user interfaces during product derivation while (2) still supporting for individual, creative design.

I. INTRODUCTION

In *Model-Driven Engineering (MDE)* we strive to build software-intensive systems with a high degree of automation and reuse, using models as the primary artifacts of construction. When developing *families* of such systems [1] we can apply techniques from *Software Product Lines (SPL)* [2], [3]. An integration of SPL and MDE facilitates the construction of families of systems with strategic reuse and a minimum of technical diversity (SPL) and a high level of automation (MDE). This has been demonstrated in a number of approaches, e.g., [4], [5].

Such highly-automated approaches work well as long as the derived artifacts can be constructed in a mechanized way on a high quality level. For the application's user interface (UI), however, a fully mechanized construction is not sufficient as it has been shown that the resulting UIs often lack of usability [6], [7], [8]. Hence, to achieve sufficient usability, UI design in practice is usually performed manually with the creativity and intelligence of human designers. The flip side is that this compromises the desired degree of automation. The situation becomes even more complex, if we do not construct one UI, but UIs for a whole product line of software applications.

As running example throughout this paper example we will use a SPL for online shops, called *OnlineShopPL*. A product of this SPL will be one particular e-commerce system tailored to the specific needs of the customer (i.e., the company running that shop). To capture the available configuration choices we can use a feature model as shown in Fig. 1 (in FODA Notation

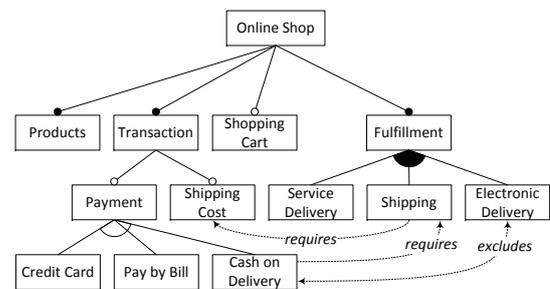


Fig. 1. Feature model of *OnlineShopPL*

[9]). To simplify, we focus on the order processing part of the online shop and abstract from other functionality.

The model specifies that a product of the *OnlineShopPL*, i.e., an online shop, has a mandatory *Products* section, an optional *Shopping Cart* and several choices on *Transactions* and *Fulfillment*. The feature model is augmented with cross-tree constraints, e.g., the selection of *Electronic Delivery* excludes the payment method *Cash on Delivery*.

To derive products from our *OnlineShopPL*, we could take a the conventional approach for model-driven product line engineering: In *Domain Engineering*, the product line is described as a feature model (describing the capabilities of the product line) and a related set of components (implementing the capabilities). In *Application Engineering*, a product is then constructed by first configuring the feature model and then deriving the product's implementation. This derivation could largely be realized by automated transformations and code generators which assemble the final product.

This approach, however, has a flip side. Such automated techniques are problematic when it comes to the creation of usable and individual UIs. If we look for real world examples of e-commerce software, which can be used to construct online shops, we will find many different installations of the same software platform. On comparing the front-ends (i.e., web sites) of the various online shops, we will find many differences in the UI and interaction design, although these installations were all built based on the same platform¹. Some variations are of purely visual nature, caused e.g., by the different shop owner's corporate identities. But there are also lots of variations in

¹For instance, see the customer reference lists of e-commerce platform providers, such as Intershop on <http://www.intershop.de/intershop/references/>

the navigation structure, the site layout, the individual UI elements, and the interaction design, depending e.g. on the kind of products presented in the shop and on the individual premises and goals of the shop owner.

Providing such highly customized UIs can be of strong importance for the shop owner as the customer experience strongly influences the online shop’s success. This holds not only for online shops but for most kind of applications which directly target the end-user [10]. Consequently, while the application core (the software components processing customers, products, transactions, payments, etc.) can be generated as described above, the UI is designed manually by UI design experts. The UI designers then have to manually ensure that the UI adheres to a given product configuration and have to manually link it to the generated application core, which are tedious and error prone tasks. Moreover, there is a conceptual gap in the development process at this point: Much effort is invested during Domain Engineering to capture knowledge precisely enough such that automated product derivation becomes possible, but none of the generated artifacts is considered (systematically) for the UI design.

This paper addresses this problem and proposes a solution which facilitates both (1) a systematic and semi-automated creation UI design during product derivation while (2) still supporting for individual, creative design. For this purpose we combine and adapt several concepts from *Model-based User Interface Development (MBUID)* and integrate them into our SPL product derivation approach. This enables to semi-automatically derive a UI during product derivation, including the connections between UI and the core application. Nevertheless, individual customization and creative design is still fully supported, but at well-defined “injection points”.

The remainder of the paper is structured as follows: We analyze UI development from the viewpoint of automation and discuss how techniques from MBUID can be used for our goals (Sec. II). We then present concepts for the integration of automated product derivation and individual UI design (Sec. III). Subsequently, we show how these theoretical concepts are put into practice in our approach for model-driven UI derivation (Sec. IV). The paper finishes with an overview of related work and conclusions.

II. AUTOMATION IN USER INTERFACE DEVELOPMENT: STATE OF THE ART

As a first step towards a solution we will now analyze available alternatives in UI development. First, we examine the overall spectrum of alternatives from the viewpoint of automation in development. We then take a closer look on the most promising alternatives, the concepts from the area of MBUID.

A. Spectrum of Available Techniques

When discussing UI development under the aspect of automation, we can consider a whole spectrum of approaches (Fig. 2). On the left-hand side we have purely manual UI

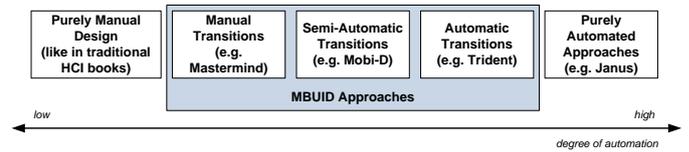


Fig. 2. The spectrum of approaches in terms of automation in UI design

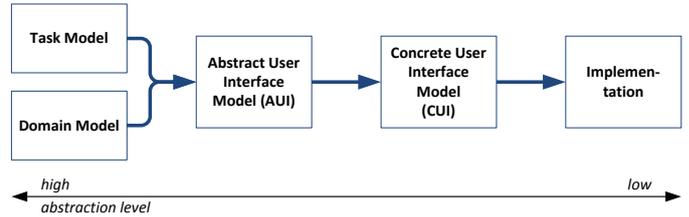


Fig. 3. Main concepts in MBUID

design without any kind of automation as described in Human-Computer Interaction (HCI) books, e.g., [11]. If applied in the context of model-driven development of the underlying application core, this means that available information is not put to proper use. For instance, the UI designers start from textual requirements, but do not systematically consider, e.g., existing feature models or domain models. The right-hand side represents the opposite extreme: a completely automated process which generates a UI from existing information. For instance, the *Janus* tool [12] can generate a UI directly from a domain model. Such fully automated approaches can only be used for very specific application domains as they often fail to provide a sufficient UI quality [6], [7], [8].

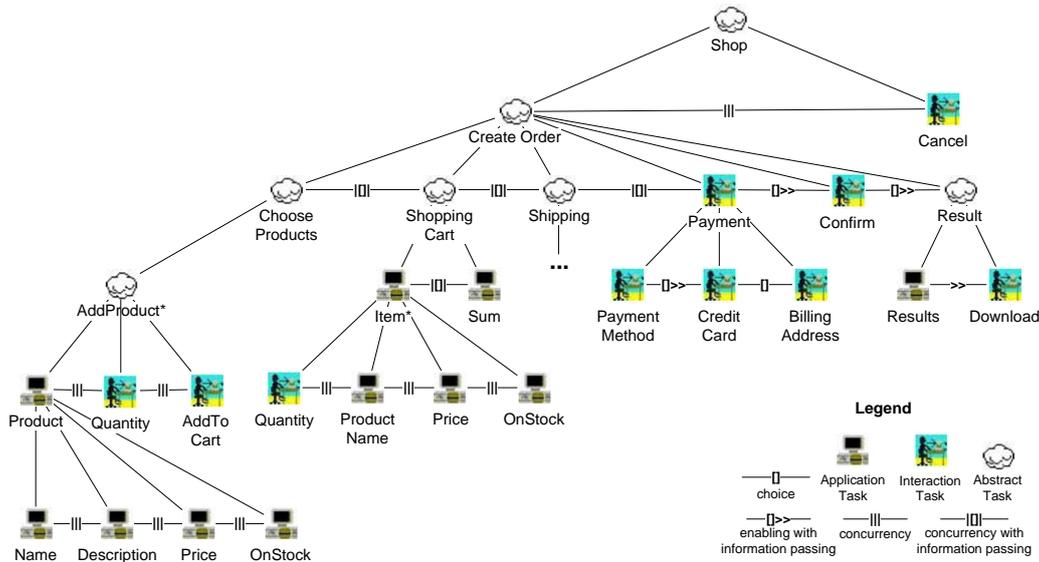
Approaches from MBUID aim to overcome these problems and to provide systematic and partially automated UI development while preserving usability. In contrast to purely automated approaches they consider additional information about the UI specified by the developers in terms of abstract models.² The following section gives a more detailed introduction into MBUID concepts, corresponding to the center part of the spectrum in Fig. 2.

B. Model-based User Interface Development (MBUID)

Model-based User Interface Development (MBUID) [6], [13], [14] can roughly be structured according to the process shown in Fig. 3: The most abstract models are a *Task Model* and *Domain Model*. The *Domain Model* specifies the structure of the application logic, e.g., in terms of a conventional UML class diagram.

1) *Task Model*: Tasks are activities performed by the user or the system to reach the user’s goals. Fig. 4 shows an example task model in *ConcurTaskTree (CTT)* notation [15], corresponding to the *OnlineShopPL* (Sec. I): An *Application Task* is performed by the system (e.g., display available products). An *Interaction Task* is performed by interaction between the user and the system (e.g., select a payment

²Here we use the term “models” in a broad sense including, e.g., XML-based description languages

Fig. 4. The Task Model for the *OnlineShopPL*

method), while an *Abstract Task* groups different types of subtasks. The horizontal lines express *temporal constraints*. For instance, a concurrent execution of tasks is shown as $|||$, a sequential execution of tasks with information passing is shown as $>>[]$.³

2) *AUI*: The *Abstract User Interface Model (AUI)* is specified based on the Task Model and the Domain Model. It describes the UI in terms *Abstract Interaction Objects (AIOs)* which are platform- and modality-independent abstractions of UI elements (widgets). (Modality-independent means that the UI is not necessarily graphical but can also be, for instance, speech-based). An example of an AIO is the *input* element which enables the user to input some data, like e.g., a text field widget. Other examples are the *output* element which present some data to the user, the *selection* element which enables the user to select a value, or the *action* element which enables the user to trigger some actions like e.g., a button. The data or the operations associated with an AIO can be specified by relationships to elements from the Domain Model.

Each AIO is related to a task in the Task Model. For instance, the task *Credit Card* could be realized by several input elements for the credit card number, card type, etc. AIOs are grouped into *Presentation Units* (abstractions of windows in a graphical UI) and other *container* elements to further structure the UI (corresponding to, e.g., Panels in Java). Until now, there is no common standard notation for AUIs models.

3) *CUI and Final Implementation*: The *Concrete User Interface Model (CUI)* realizes the AUI for a specific modality in terms of concrete widgets and layout. Like for the AUI, there is no standard CUI notation. Subsequently, the final UI implementation is generated, usually under consideration of information from all other models. Depending on the purpose, many approaches use additional models, e.g., a Context Model

for context-sensitive UIs.

4) *Automation within MBUID*: Within the UI modeling there are still different degrees of automation (see [6], [16]) as illustrated by the center part of Fig. 2. For instance, one can provide model transformations for an automatic transition from Task and Domain Model to the final implementation, like in *Trident* [17], or require the modeler to manually specify all UI models and the relationships between, like *Mastermind* [18].

For our purpose, semi-automatic approaches, like *Mobi-D* [7] seem to be most promising. They aim to provide as much automation as possible while enabling manual customization for critical decisions. Typical critical decisions are (1) the decomposition of the UI into presentation units (e.g., whether to put the shopping cart on a separate screen or joined with the product selection) and (2) the mapping of AIOs to CIOs (e.g., whether a selection element is mapped to a list box or to a drop down list) [6], [7], [16].

III. INTEGRATING AUTOMATED PRODUCT DERIVATION AND INDIVIDUAL USER INTERFACE DESIGN

Based on the preceding analysis, the following section discusses two resulting general principles for derivation of UIs: (1) *Derivation on an abstract level* and (2) *Integration of customization into an automated process*.

A. Derivation of Abstract User Interface Models

From a technical point of view, the UI is just another software subsystem. Consequently, a straightforward application of product derivation concepts to the UI would work as follows: During Domain Engineering, the UI for the complete SPL is constructed and the required UI components are implemented. During product derivation, the required components are selected according to the configuration and automatically composed to form the product's UI. From the viewpoint of

³See [15] for a complete description of CTT model element types.

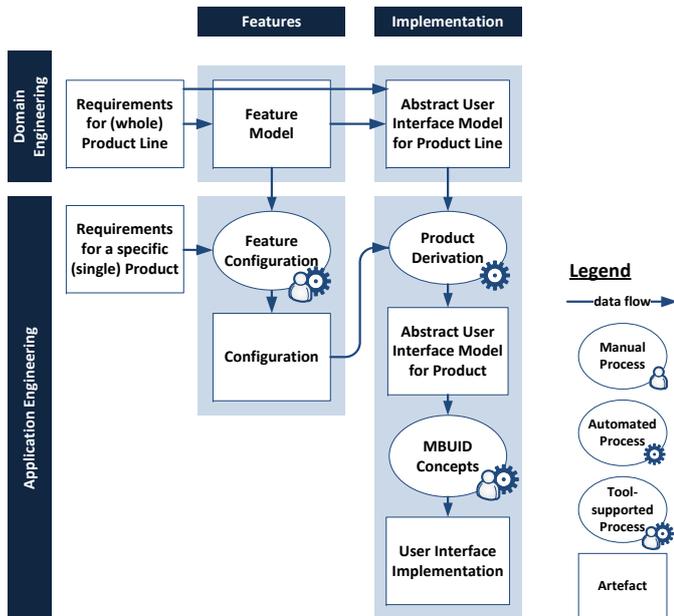


Fig. 5. Basic concept for semi-automatic derivation of user interfaces

UI development, this naive solution would correspond to a purely automated approach without considering human design knowledge or customization, which is insufficient for most cases. Thus, product derivation for UIs requires a different approach, considering the UI-specific body of knowledge from MBUID.

Hence, we propose to perform the UI derivation at a *higher level of abstraction* using the models from MBUID. Indeed, our example models (Figures 1 and 4) show that a Task Model could be related to a feature model: For instance, the task CreditCard corresponds to the feature Credit Card, the task BillingAddress to the feature PayByBill, the task PaymentMethod depends on the selected Payment features, etc. Of course, one feature can be related to multiple tasks and vice versa; for instance, selecting the feature Electronic Delivery indicates not only that the tasks related to keywordShipping can be omitted but also the task OnStock.

As the AIOs are associated with tasks, they can be (indirectly) related to features as well. Thus, it is possible to automatically determine which AIOs are required for a given product configuration. Other UI decisions should be optimized manually, e.g., the decomposition into Presentation Units and the mapping onto concrete UI elements. These decisions strongly depend on the details of the concrete configuration and product context. For instance in the *OnlineShopPL* we have to consider the type of items sold in the shop, as this influences form of product presentation and selection (and e.g., the space required on the UI).

By combining SPL and the concepts from MBUID (Fig. 3) we come to the integrated approach we propose for derivation of UIs (see Fig. 5): Domain Engineering processes at an abstract level, i.e., with a model of the Abstract User Interface. During Application Engineering, the product-specific AUI

can be calculated automatically from the product's feature configuration. On that base, the final UI is derived using semi-automatic approaches from MBUID. Some parts of the UI implementation, like the links between UI elements and application logic, can be generated fully automatically. Others, like the concrete layout, the visual appearance, and the selection of concrete UI components are generated automatically, but can be customized if desired.

This general framework introduced here still abstracts from the concrete modeling languages used for its realization. For instance, it is necessary to select adequate UI models and transformations from the various MBUID approaches. We will show a possible realization in Sec. IV.

B. Systematic Integration of UI Customization Techniques

So far, we have argued that the step from the AUI model to the final implementation must include the option for manual customization by the designers. We will now analyze potential alternatives and introduce two techniques which contribute to our approach.

In general, customization of models and model transformations is a common task in model-driven engineering. Given one transformation one can customize it for instance by 1) adding additional information to the source model (e.g., tagged values), 2) tuning the transformation itself (e.g., by specifying parameters [19]), or 3) by just post-editing the resulting target model. These basic possibilities are used in MBUID [16] and could be used in our approach.

However, it is desirable to provide additional, more specific customization techniques since, 1) during product derivation efficiency is more important than maximal flexibility and 2) the UI designers might not be familiar with generic modeling tools and transformation languages and require more domain-specific (i.e., UI-specific) tool support during customization. An important example in this sense is *MOBI-D* [7], which provides two concepts for customization. During the transformation, the designer is provided with a dialog where he can adjust parameters in a graphical UI. For customizing the CUI, it provides a specific kind of GUI Builder where the designer can select from those CIOs which correspond to the AIOs from the AUI.

In the following we propose two customization techniques for our purpose which enables the designer full control about all aspects of the UI while enabling a high degree automated tool support: *Tree-based UI Clustering* and the *UI Placeholder* concept.

1) *Tree-based UI Clustering*: Our approach for tree-based UI clustering addresses the problem of decomposing the UI into presentation units. It is based on earlier work [20], we adapt it here to handle specific requirements of product variability. In contrast to other approaches, the approach uses information from both Task Model and AUI and represents them in a common view. Fig. 6 shows this for our *OnlineShopPL*: The basic tree structure represents the AUI in terms of a hierarchy of AIOs. The AIO types are mainly those mentioned in Sec. II-B with additional support for media

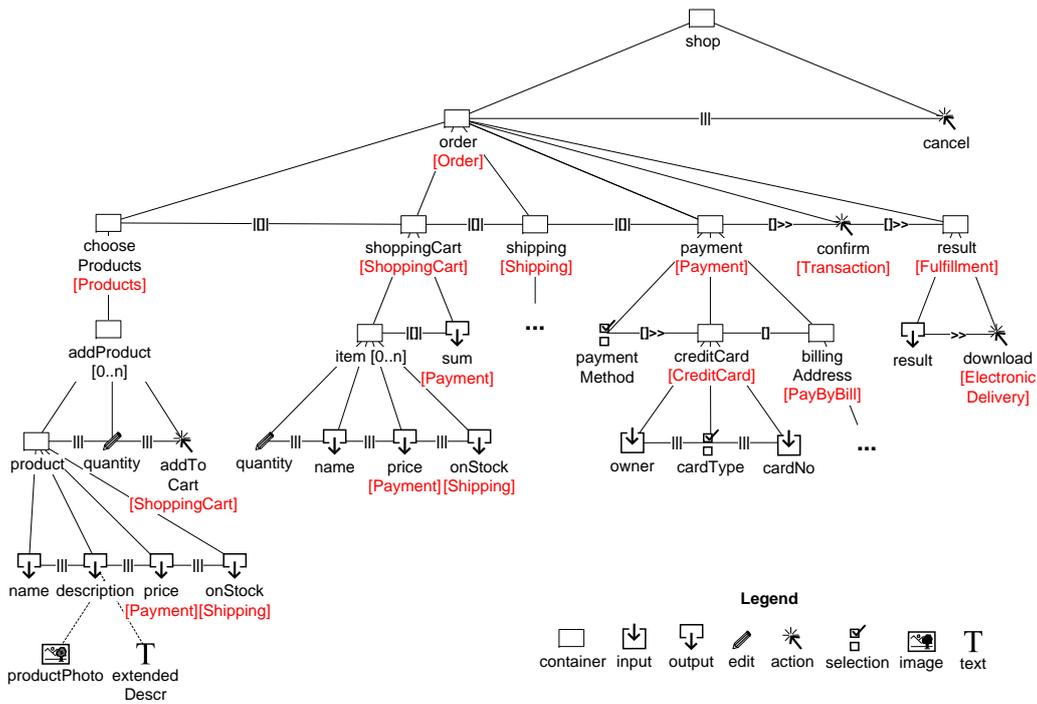


Fig. 6. The Abstract UI as a tree hierarchy for defining the Presentation Units

objects like image, text, 3D graphics, etc. The annotations in square brackets show the corresponding features and will be discussed later in Sec. IV-A. The horizontal lines specify temporal constraints derived from the Task Model. Based on these information, a heuristic calculates the clustering of AIO into Presentation Units. The heuristic can be influenced by parameters to take into account the requirements of a particular CUI platform (e.g., smaller presentation units for a mobile interface).

Fig. 7 shows the result of the clustering for two different AUIs (AUI+C). Even if the designer wants to manually customize the clusters, the automated heuristic still provides a starting point which is very helpful when dealing with UIs with many elements. In addition, based on the AUI+C representation it is possible to create interactive visual tools which enable the designer to modify the clustering very easily e.g., by drag and drop.

2) *The User Interface Placeholder Concept*: The placeholder concept, generalized from earlier work [21], addresses the development of the CUI based on the clustered AUI, i.e., the choice of AIOs, their concrete visual appearance, and the layout within the presentation units. The basic idea is to transform the AUI to generate a skeleton implementation, which can be modified in UI-specific visual authoring tools, e.g., the multimedia authoring tool *Flash*. In particular, the generated UI is composed of *placeholders* which can be customized and refined.

By using an unique identifier (ID) for each placeholder we can automatically associate it with corresponding elements, e.g., links to the application logic or event handling code gen-

erated from the model. Moreover, we can trace placeholders, independent from the designers modifications. As long as the designer does not delete the placeholder itself, all generated information remains untouched while the designer can freely use all the authoring tool's powerful visual functionalities without any further restrictions.

IV. DETAILED REALIZATION

In the preceding section we took first steps towards a solution by developing some principles on a conceptual level. We will now present an approach that integrates this concepts and puts them into practice. Figure 8 shows our detailed approach which we explain in the following step by step using our *OnlineShopPL*.

A. Domain Engineering

In *Domain Engineering* (upper layer in Fig. 8) we perform the processes 1 to 5 to create and describe the product line in terms of five models, **A** to **E**.

The process starts with **Feature Analysis** 1, which analyses the *Product Line Requirements* and produces the *Feature Model* **A**, capturing the scope and capabilities of the product line. The Feature Model for our *OnlineShopPL* was shown earlier in Fig. 1.

The next step is **Task Analysis** 2, which takes into account the *Product Line Requirements* and the *Feature Model* to create a *Task Model* **C**. The task model for our *OnlineShopPL* was presented in Fig. 4 in CTT notation.

After completing the Task Model this is turned into an **Abstract User Interface** (AUI) Model in two steps. First, the

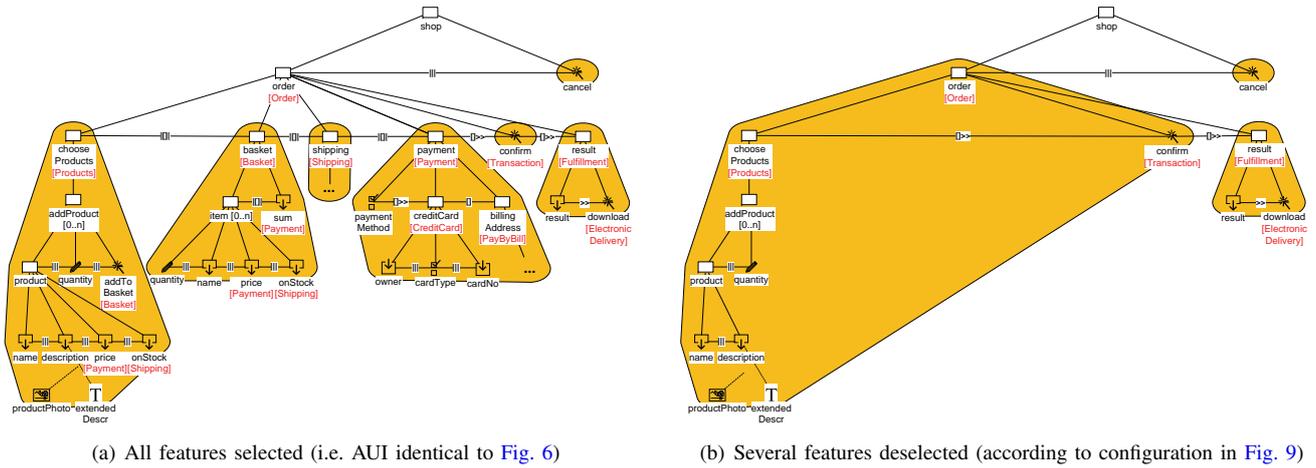


Fig. 7. Clustered AUI Models (AUI+C) for two different feature configurations

Task Model is transformed into the *AUI Model* **E** using an automated *Refinement Transformation* **3**, which also remembers the links between tasks and AUI elements, which were derived from these tasks, by creating a *Mapping Model* **D**. Second, the engineer can perform *Manual Adjustments* **4** to further refine the AUI model.

The final activity within Domain Engineering is *Feature Mapping* **5**, where features are connected to the corresponding elements in the *Task Model* and the *AUI Model*. This information is stored in a second *Mapping Model* **B**. Hence, as a final result of all these steps we get an abstract UI model, whose elements are mapped onto the corresponding features (via Mapping Model **B**) and tasks (via Mapping Model **D**). The AUI for our *OnlineShopPL* was shown earlier in Fig. 6. The red annotations in square brackets show the mappings to the corresponding features. It should be noted that some features influence multiple locations of the UI, similar to cross-cutting concerns in aspect-oriented programming. For instance, consider the feature Payment. If payment is deselected (e.g., because customers get individual offers via mail or email), this not only removes elements for specifying the payment method, but also influences the presentation of products and the shopping cart.

B. Application Engineering

After the SPL has been established we can start *Application Engineering* (see lower layer in Fig. 8), where we perform the processes **6** to **11** to derive products.

The derivation of products from the product line starts with *Feature Configuration* where we try to match *Product Requirements* with the product line's capabilities as described in the *Feature Model*. This results in a *Product Configuration* **A₃** (or the insight that some requirements are not covered by the product line). Figure 9 shows an example of a feature configuration for an online shop, with a minimal set of features selected.

With the decisions captured in the *Feature Model Configuration* we can now perform *Product Derivation*, where we

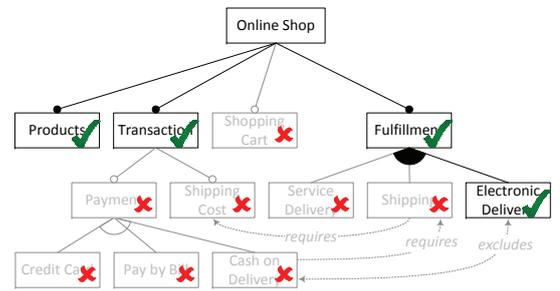


Fig. 9. A possible feature configuration for a concrete product

use a technique called *Negative Variability* [5], which works as follows: The models on product-line level (**C** and **E**) contain the union of all potential model variations. Based on the *Feature Model Configuration* **A₃** and the feature-implementation mappings **B** we can filter out elements (i.e., tasks and AUI elements) which are related to eliminated features and, hence, are not required for this particular product. As a result we get the product-specific *Task Model* **C** and *AUI Model* **E**, as well as *Mappings* **D** between them. These are subsets of the product-line models shown earlier (see figures 4 and 6), with all elements removed that correspond to eliminated features.

Given the derived task and AUI model we can perform *Clustering* **7** to determine Presentation Units, which later will become, e.g., screens or forms. For this step we use the clustering technique from Sec. III-B. The designer can either customize the clustering based on the tree hierarchy or proceed with the generated model and, if required, modify the Presentation Units later in the authoring tool (step **11**). Figure 7(b) discussed earlier, represents the clustered AUI (AUI+C) for the minimal configuration in Fig. 9.

Finally, we apply a model transformation *AUI-to-CUI* **9** to turn the AUI+C model into an platform-specific *CUI Model* **C**, which is then fed into an *CUI Generator* **10** to create the *CUI Implementation* **I**. We are experimenting with multiple AUI-to-CUI generators for different platforms (e.g., GUI, Web,

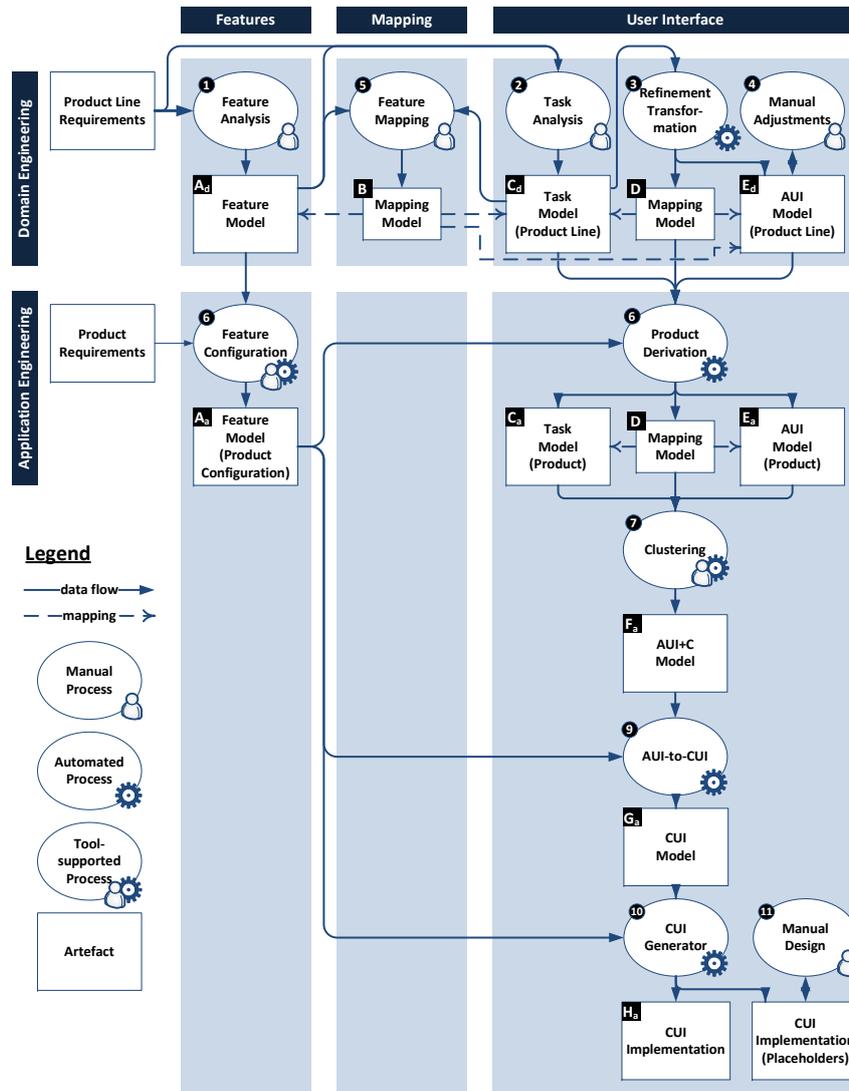


Fig. 8. Overview of the process

mobile). These have been omitted from the illustration in Sec. IV. As an example, Fig. 10 shows the generated UI skeleton for the Adobe Flash UI platform.

The CUI generators use the placeholder concept from Sec. III-B. This means that (1) the UI structure and the glue code for the integration with other subsystems are completely generated from the model and (2) design related artifacts (which determine visual elements, layout, appearance) are generated as placeholders. In *Manual Design* 11 these placeholder can be directly loaded into authoring tools, such as Adobe Flash, where the UI designer can use the functionality of professional design software and all possibilities of their human creativity.

V. CONCLUSION

For many applications, an *individually* designed, usable, and aesthetic user interface is a key success factor. On the other hand, *automated* approaches with techniques from model-

driven development and software product lines promise improvements in time to market, cost, productivity and quality [2].

In this paper we address the integration of these, initially contradictory, goals. To this end, we have carefully considered the related work and approaches from UI development, MBUID in particular, and gathered a systematic overview from the viewpoint of automation (Sec. II). Regarding SPL, we build up on the state-of-the-art for model-driven product derivation, as discussed in Sec. I. Until now, the few existing approaches which try to integrate UI construction with product derivation (e.g., [22]), use the straightforward derivation approach described in Sec. III-A, which is only suitable for very specific UIs without customization. To the best of our knowledge, there is no approach yet that integrates automated product derivation and individual UI design as addressed in this paper.

We have elaborated two general concepts, the derivation on the level of abstract UI models, and the systematic inte-

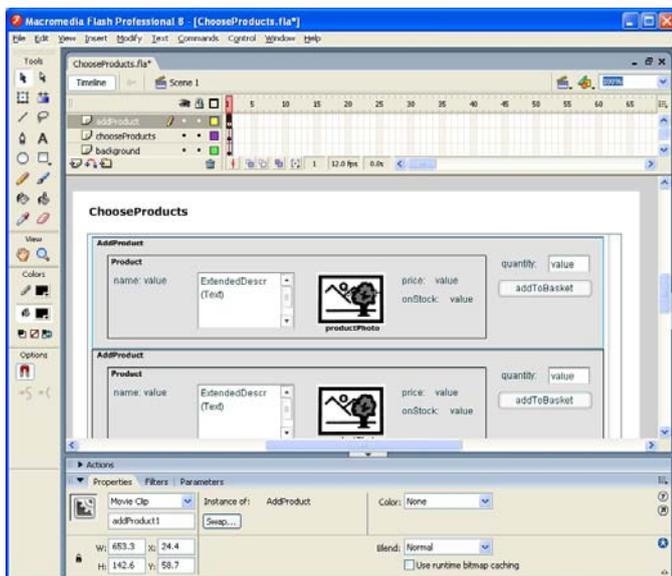


Fig. 10. The generated user interface skeleton for the screen ChooseProducts based on the AUI model in Fig. 7(a)

gration of UI customization techniques (Sec. III). Moreover, we have shown a concrete realization of these concepts by a concrete, detailed process illustrated by the online shop example (Sec. IV).

The different steps within our process are supported by prototypical tools and model transformations. We use various Eclipse-based frameworks like the Eclipse Modeling Framework (EMF), GMF (Graphical Modeling Framework), oAW (openArchitectureWare), and ATL (ATLAS Transformation Language). Based on these we implemented a tool chain for feature configuration ([23], [24]) and product derivation (using negative variability, see Sec. IV-B). The clustering and further processing of interaction elements is implemented as an ATL transformation adapted from earlier work [20]. The code generation including the placeholder concept reuses the ATL transformations from [21].

Using our concepts, large parts of UI development are performed automatically: The selection of the (abstract) UI elements according to the product configuration, the implementation of the UI's overall structure, and the implementation of relationships of UI elements is derived automatically from the product configuration. For all parts with need for custom design (like the selection of the concrete UI elements, UI layout, and the concrete visual appearance) we provide systematic support by generating a consistent starting point (including all required relationships) which can then be customized and refined visually.

Future work includes a more detailed evaluation and the gathering of experience with the approach, which is currently still on a conceptual prototype level. In particular, we intend to explore support for traceability and (iterative) product evolution. Another potential research direction is the combination with MBUID approaches for context-sensitive UIs (e.g., [25]) by considering the product configuration as a specific kind of

context. Finally, we plan to analyze variability in UIs more systematically. A question is, for instance, which variations between UIs are rather accidental or caused by the designer's personal preference and which cause a measurable difference in usability.

ACKNOWLEDGMENTS

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre, <http://www.lero.ie/>.

REFERENCES

- [1] D. Parnas, "On the design and development of program families," *IEEE Trans. Softw. Eng.*, vol. 2, no. 1, pp. 1–9, March 1976.
- [2] P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley, 2002.
- [3] K. Pohl, G. Boeckle, and F. van der Linden, *Software Product Line Engineering*. New York, NY: Springer, 2005.
- [4] D. Dhungana, R. Rabiser, P. Grünbacher, and T. Neumayer, "Integrated tool support for software product line engineering," in *ASE '07*, 2007.
- [5] M. Voelter and I. Groher, "Product line implementation using aspect-oriented and model-driven software development," in *SPLC'07*, 2007.
- [6] P. A. Szekely, "Retrospective and challenges for model-based interface development," in *DSV-IS*, 1996.
- [7] A. Puerta and J. Eisenstein, "Towards a general computational framework for model-based interface development systems," in *IUI*, 1999.
- [8] B. A. Myers, S. E. Hudson, and R. F. Pausch, "Past, present, and future of user interface software tools," *ACM Trans. Comput.-Hum. Interact.*, vol. 7, no. 1, pp. 3–28, 2000.
- [9] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature oriented domain analysis (FODA) feasibility study," SEI Technical Report CMU/SEI-90-TR-21, 1990.
- [10] S. Krug, *Don't Make Me Think: A Common Sense Approach to the Web*, 2nd ed. Thousand Oaks, CA, USA: New Riders Publishing, 2005.
- [11] B. Shneiderman and C. Plaisant, *Designing the User Interface*, 4th ed. Addison Wesley, 2004.
- [12] H. Balzert, F. Hofmann, V. Kruschinski, and C. Niemann, "The janus application development environment," in *CADUI*, 1996.
- [13] P. P. da Silva, "User interface declarative models and development environments: A survey," in *DSV-IS*, 2000.
- [14] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, N. Souchon, L. Bouillon, M. Florins, and J. Vanderdonck, "Plasticity of user interfaces: A revised reference framework," in *TAMODIA*, 2002.
- [15] F. Paternò, C. Mancini, and S. Meniconi, "Concurtasktrees: A diagrammatic notation for specifying task models," in *INTERACT'97*, 1997.
- [16] T. Clerckx, K. Luyten, and K. Coninx, "The mapping problem back and forth: customizing dynamic models while preserving consistency," in *TAMODIA*, 2004.
- [17] J. M. Vanderdonck and F. Bodart, "Encapsulating knowledge for intelligent automatic interaction objects selection," in *CHI '93*, 1993.
- [18] P. A. Szekely, P. N. Sukaviriya, P. Castells, J. Muthukumarasamy, and E. Salcher, "Declarative interface models for user interface construction tools: the mastermind approach," in *EHCI*, 1995.
- [19] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Syst. J.*, vol. 45, no. 3, pp. 621–645, 2006.
- [20] G. Botterweck, "A model-driven approach to the engineering of multiple user interfaces," in *MoDELS Workshops*, 2006.
- [21] A. Pleuß and H. Hußmann, "Integrating authoring tools into model-driven development of interactive multimedia applications," in *HCI Int.*, 2007.
- [22] K. Garcés, C. Parra, H. Arboleda, A. Yie, and R. Casallas, "Variability management in a model-driven software product line," *Rev. Av. en Sist. e Informática*, vol. 4, no. 2, 2007.
- [23] G. Botterweck, M. Janota, and D. Schneeweiss, "A design of a configurable feature model configurator," in *VAMOS 2009*, 2009.
- [24] G. Botterweck, D. Schneeweiss, and A. Pleuß, "Interactive techniques to support the configuration of complex feature models," in *MDPLE*, 2009.
- [25] J. Van den Bergh and K. Coninx, "Cup 2.0: High-level modeling of context-sensitive interactive applications," in *MODELS*, 2006.

Supporting Stepwise, Incremental Product Derivation in Product Line Requirements Engineering

Reinhard Stoiber, Martin Glinz

Department of Informatics, University of Zurich, Switzerland

Email: {stoiber, glinz}@ifi.uzh.ch

Abstract—Deriving products from a software product line is difficult, particularly when there are many constraints in the variability of the product line. Understanding the impact of variability binding decisions (i.e. of selecting or dismissing features) is a particular challenge: (i) the decisions taken must not violate any variability constraint, and (ii) the effects and consequences of every variability decision need to be understood well. This problem can be reduced significantly with good support both for variability specification and decision making. We have developed an extension of the ADORA language and tool which is capable of modeling and visualizing both the functionality and the variability of a product line in a single model and provides automated reasoning on the variability space.

In this paper we describe how our approach supports stepwise, incremental derivation of a product requirements specification from a product line specification. We visualize what has been derived so far, automatically re-evaluate the variability constraints and propagate the results as restrictions on the remaining product derivation options. We demonstrate our approach by showing a sequence of product derivation steps in an example from the industrial automation domain. We claim that our approach both improves the efficiency and quality of the derivation process.

I. INTRODUCTION

Product derivation is the process of defining a single application product based on a product line variability model. The product derivation process begins with a product line domain variability model, where all variability is unbound and ends with a single concrete product model. During this process, all product line variability needs to be bound, i.e. selected or dismissed.

Product line variability models can become very complex, particularly when the variability is restricted by many constraints. When deriving a product from a product line, an application engineer is challenged with two problems: (i) the decisions he or she takes must not violate the variability constraints, and (ii) he or she needs to understand the effects and consequences of every variability binding decision in order to derive a high-quality product configuration.

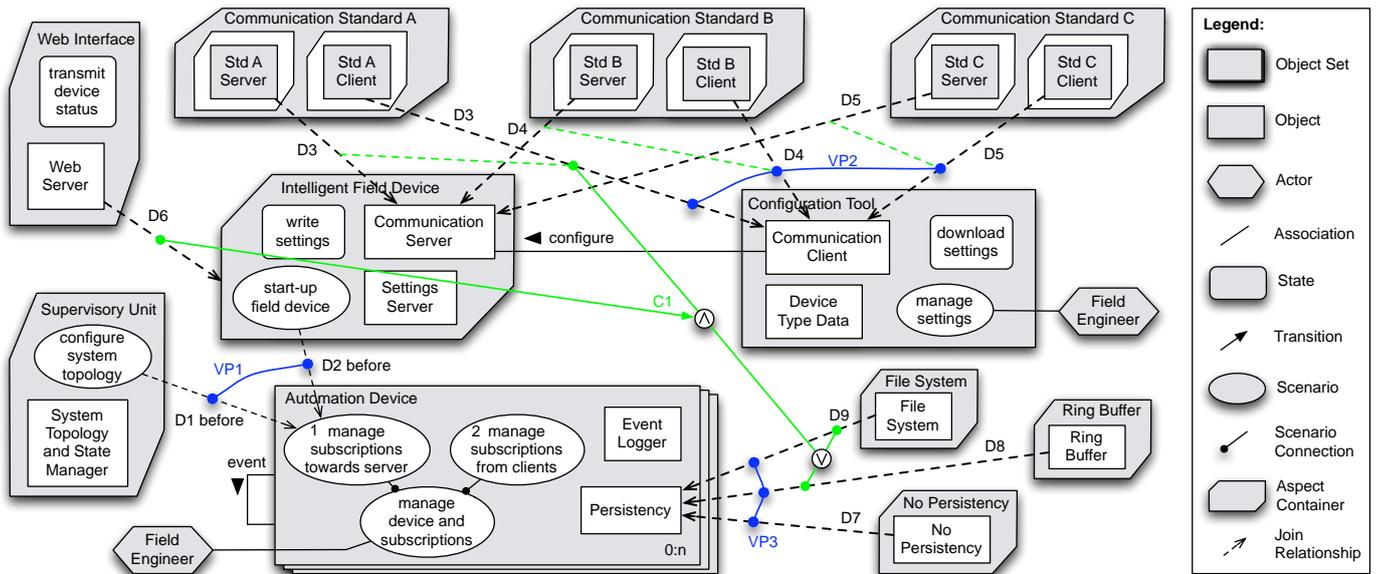
Existing solutions for product derivation from a product line variability model mostly build upon feature-oriented domain analysis [1] or slightly extended versions thereof. However, feature modeling languages have two significant limitations. First, feature models can only define rather simple variability constraints. Schobbens et al. [2] conducted a survey of feature modeling languages that claimed to improve the expressiveness of the original FODA method [1]. According to this survey, current feature modeling languages support only simple

types of constraints, mostly based on cardinalities, for example and/or/xor relationships between sub-features with the same parent feature, and *requires*, *excludes* or *mutual exclusion* dependencies between features. For complex constraints, this is too simplistic. For example, a constraint such as $F1 \text{ implies } F2 \text{ or } F3$ cannot be specified with a single constraint in a feature model. Second, feature models provide only the names of the available features, but no detailed information about their concrete functionality. In order to comprehend the meaning and impact of a feature, additional documentation or domain expert knowledge is required. Mannion [3], Jarzabek et al. [4] and Czarnecki et al. [5] [6] have addressed this limitation by introducing mappings between features and single textual requirements, requirements documents and UML models, respectively. Thus, with appropriate tool support, the detailed impact of a feature can be shown in other documentation, but this requires a lot of context switching, which is not ideal.

We have developed a new approach to product line requirements modeling that addresses and aims to solve these two problems. Our approach builds on the graphical object-oriented requirements specification language ADORA [7], ADORA's aspect-oriented modeling capabilities [8] which we use for modularizing product line variability, and a new table-based boolean decision modeling concept which we use for managing the product line variability. Using aspects for separately modularizing variability (i.e. variable features) and its composition semantics allows us integrate the variability model into the graphic requirements model and visualize them together.

In this paper, we concentrate on describing our approach to variability decision and constraints modeling and how we can support stepwise, incremental derivation of products. The main idea is to encode the constraints in tables with boolean logic. This allows us to apply existing boolean satisfiability (SAT) solving tools for automated reasoning and verification during product derivation. Using an industrial example, we demonstrate how our techniques enable a stepwise and incremental approach to product derivation.

The remainder of the paper is organized as follows. In Section 2 we introduce an industrial product line requirements specification as a running example and briefly explain the ADORA product line modeling approach. In Section 3 we motivate and describe how we support stepwise and incremental product derivation. In Section 4 we demonstrate the application of our approach to the example introduced



Decision Table

ID	Description / Derivation Question	Design Rationale	Constraints	ifTrue	ifFalse	Decision
D1	Should the automation device be a supervisory unit?	The automation device platform is used for cost and quality reasons.	VP1	-D2	D2	undecided
D2	Should the automation device be an intelligent field device?	The automation device platform is used for cost and quality reasons.	VP1	-D1	D1 -D6	undecided
D3	Should the intelligent field device support communic. std. A?	Standard A is the newest and most performant one.	VP2, C1		-D6	undecided
D4	Should the intelligent field device support communic. std. B?	Is a must in some countries.	VP2			undecided
D5	Should the intelligent field device support communic. std. C?	C is still required by many legacy systems.	VP2			undecided
D6	Should the intelligent field device a web interface?	The web interface allows access over the world wide web.	C1	D3 -D7		undecided
D7	Is there no memory in the automation device?	Might be interesting for non-critical systems.	VP3	-D6 -D8 -D9		undecided
D8	Is there a ring buffer in the automation device?	Ring buffers are cheap to implement and fast.	VP3, C1	-D7 -D9		undecided
D9	Is there a file system in the automation device?	File system offer huge amounts of storage space.	VP3, C1	-D7 -D8		undecided

Variation Points Table

ID	Description / Rationale	minCard	maxCard	Decisions Involved
VP1	An automation device always can be either a supervisory unit or an intelligend field device.	1	1	D1, D2
VP2	There are three different communication standards to configure a field device: standard A, B, and C. At least one must be chosen.	1	3	D3, D4, D5
VP3	There exist three different persistency types: no persistency, a ring buffer, or a file system. One of these must be selected.	1	1	D7, D8, D9

Constraints Table

ID	Description / Rationale	Antecedent	Operator	Consequent
C1	A web interface always requires the communication standard A and a local persistency installed (either ring buffer or FS).	D6	=>	D3 and (D8 or D9)

Fig. 1. An automation device product line specified in the ADORA language.

above. Section 5 discusses scalability issues, related work and concludes.

II. PRODUCT LINE DOMAIN MODELING: AN INDUSTRIAL EXEMPLAR

To demonstrate our approach, we employ a product line requirements model of industrial automation devices at the product and components level [9]. Fig. 1 shows the graphical requirements specification and the decision and constraints tables of this product line. The graphical notation is briefly explained in the legend on the right-hand side of the figure. The commonality is modeled with abstract objects or object sets and the variable requirements are modularized with aspect containers and graphical join relationships. Every variable join relationship is annotated with a decision item. The details of these decision items are modeled in tables. The subsequent description gives a very brief overview only. For more details, see [9].

A. Overview of the example

In our product line example, which is given in Fig. 1, the commonality of the product line consists of two components: a set of automation devices and a configuration tool. The variability is constituted by the following nine variants: there are two alternatives for the realization of an automation device: a supervisory unit and an intelligent field device. For an intelligent field device, an optional web interface may be added. For the persistency of an automation device, there are three alternatives. Finally, up to three different communication standards for supporting the configuration of an intelligent field device by a configuration tool may be chosen.

In our variability model, every variant is described in detail by the model elements given in its aspect container. The structure of the variability (equivalent to the structure in a feature tree, but only modularizing the *variable* features) is given by join relationships from the variants to the model elements where the variants apply. Every join relationship is annotated with a boolean decision item. These decision items and the constraints applying to them are modeled in tables as

follows.

The *Decision Table* lists all the decisions of the graphic variability model. The columns of the table provide detailed information about decisions, such as a description or constraints. The rightmost column is used to record decisions actually taken in the product derivation process. Initially, in domain modeling, every decision item is undecided by default. Variation points are specified in the *Variation Points Table* with their cardinalities and the decision items involved. Finally, cross-tree constraints that cannot be expressed just as variation points are specified in the *Constraints Table*. Such constraints can be arbitrarily complex formulas in boolean logic. As Fig. 1 shows, ADORA is also capable of visualizing variability constraints graphically in the requirements model [10]. For example, the solid line connecting the join relationship arrows labeled D1 and D2 in Fig. 1 visualizes that these two decision items constitute variation point VP1.

Yet undecided variability is visualized as aspects, according to Fig. 1. As soon as a variability binding decision is taken to true, the corresponding variability is woven into the model at the point(s) designated by the join relationship(s) or, if the decision was taken to false, removed from the model (see Fig. 4).

B. Automated Constraints Analysis

When working with many constraints, the modeler needs support for determining how the constraints impact his or her freedom to take further variability binding decisions. Particularly, he or she needs to know whether a set of decisions is compatible and how taking a certain decision influences the constraints for the remaining, yet undecided decision items. The ADORA tool is capable of providing such support by analyzing the constraints and checking the satisfiability of all currently interesting decision configurations using a boolean satisfiability (SAT) solving tool. The resulting constraint propagations are listed in the columns *ifTrue* and *ifFalse* in the *decision table*. Thus, a modeler can immediately see the consequences of any variability binding decision he or she takes. Moreover, we can guarantee that a partially or fully derived product always satisfies all constraints. This is very similar to Don Batory's idea of building a logic truth maintenance system (LTMS) [11]. In our solution, we additionally calculate constraint propagation previews as follows.

To calculate the constraint propagations, we iterate over all currently undecided decision items in the decision table and decide them once true and once false. For every of these partial configurations, we again iterate over all remaining, still undecided decisions and decide them again once true and once false. All resulting configurations are checked whether or not they satisfy all constraints, using a SAT solver. If a configuration is satisfiable, no propagation is needed. Otherwise, we know that the combination of two decision items violates the constraints. In this case, we record a constraint propagation: if the first decision of the violating configuration was true, we enter the negation of the second decision item into the *ifTrue* column of the first decision item, or, if the first decision was

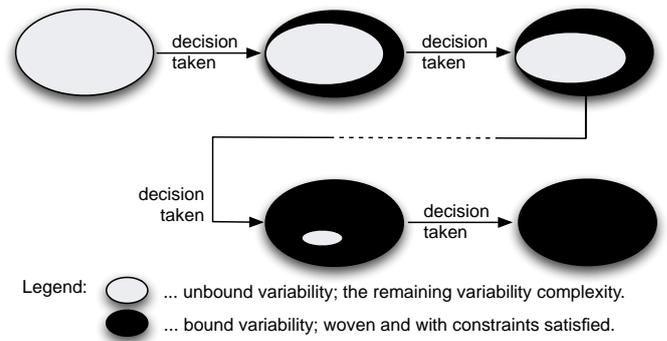


Fig. 2. Stepwise and incremental product derivation in ADORA; with every new variability binding decision the remaining product line variability gets reduced and simpler until a valid final product is derived.

false, into the *ifFalse* column. The propagation of constraints is transitive, so for every constraint propagation, we also have to calculate all transitively triggered propagations. This process continues until all potential decisions have been evaluated.

Whenever a decision is actually taken, all constraint propagations are executed. For example, if we decide decision item D2 in the model of Fig. 1 to false, we consequently must decide D1 to true and D6 to false in order to derive a product that satisfies all constraints.

Although SAT solving can be rather computation-intensive, we have not yet experienced any major performance problems. An in-depth performance analysis is subject to future work.

III. STEPWISE AND INCREMENTAL PRODUCT DERIVATION

A. Motivation

A product line domain model contains the full variability and all the variability constraints. Deriving a product configuration (in our case a requirements specification for an individual product) from such a domain model in a single step is almost impossible for any real-size product line: too many options and constraints have to be considered all at once. Hence it is rather straightforward to employ a stepwise process where variability binding decisions are taken one at a time. Fig. 2 illustrates such a process.

However, any mistake in a stepwise product derivation process makes all subsequent steps invalid and, hence, useless. Typical mistakes include decisions that yield an inconsistent configuration or lead into a dead end (i.e. a configuration that can't be further evolved towards the desired product). Consequently, a stepwise product derivation process requires sophisticated tool support, particularly for ensuring consistent intermediate configurations and for analyzing the impact of variability binding decisions, with respect to (i) the effects that the chosen or dismissed variability has on the final product requirements specification, and (ii) the extent to which the current decision restricts the options for the remaining decisions. Furthermore, a tool should also support reverting already taken decisions.

In the subsequent subsection, we describe how we provide such support in ADORA.

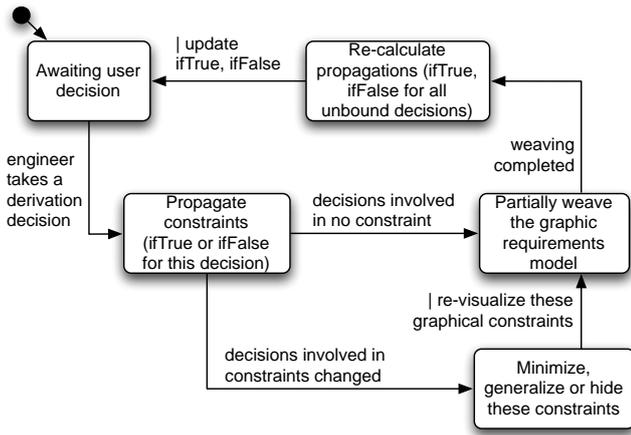


Fig. 3. A statechart describing the behavior of the ADORA tool when executing a variability binding decision taken in a product derivation process.

B. Stepwise and Incremental Product Derivation with ADORA

Product derivation in ADORA is an incremental, step-by-step process. Every time a new variability binding decision is taken or an already taken decision is reverted, the ADORA tool (i) executes the constraint propagations that are associated with this decision, (ii) adapts and re-visualizes all involved graphic variability constraints, (iii) partially weaves the graphic model to reflect the new variability configuration, and (iv) re-calculates the constraint propagations for all still unbound variability decision items (Fig. 3). The derivation process continues until all variability is bound and a concrete product requirements specification has been defined. Note that our current implementation supports forward decision making only. Support for reverting decisions is a part of our ongoing research.

In the remainder of this section, we explain the tasks executed by the ADORA tool in a derivation step (Fig. 3) more in detail. Note that this procedure is fully automatic.

When an engineer takes a decision in the derivation process, the corresponding decision item in the decision table (cf. Fig. 1) is set to true or false. The decision is recorded in the *Decision* column of the decision table. ADORA now executes the tasks described in Fig. 3.

The first task is to propagate the constraints: depending on the truth value of the decision, the decisions listed in the *ifTrue* or in the *ifFalse* columns of the decision table are taken automatically. These decisions transitively propagate their constraints in the same way. All taken decisions are recorded in a decision history. This allows undoing constraint propagations when a decision is undone.

Taking a decision may affect the graphic visualization of variability constraints. For example, if a constraint states that at least one of three options must be selected and a decision is taken that selects one of these options, the constraint is satisfied and no longer needs to be visualized. Therefore, after taking a decision, ADORA needs to *minimize* constraints that are partially satisfied by the decision taken and *hide* those

that are fully satisfied. On the other hand, when a taken decision is reverted, the graphic constraint visualizations need to be *generalized* (i.e. restored to their previous state). As the latter is not yet implemented, we focus on the techniques for minimizing and hiding constraints in this paper.

For cardinality based constraints (as listed in the variation points), the minimization of constraints is rather easy and straightforward. For example, if in Fig. 1 one of the decision items D7, D8 or D9 is taken with false, then the variation point VP3 will still be a restriction for the remaining variability configuration space and thus still needs to be displayed for the remaining two decision items. If, on the other hand, one of these three decision items is decided to true, then the other two decision items need to be set to false due to constraint propagations. As a consequence, the VP3 constraint is fully satisfied and will not be visualized any longer.

For the other constraints (as listed in the constraints table), the adaptation is more difficult, since these constraints (i) are defined as implications and (ii) may be arbitrarily complex boolean logic formulas that can span over a large set of decision items. In our example shown in Fig. 1, constraint C1 is such a non-trivial cross-tree constraint. If all involved decision items for such a constraint are undecided and one of these decision items gets bound, then ADORA automatically checks if the constraint is still only partially satisfied, and thus needs to be minimized, or if the constraint is already fully satisfied and thus needs to be hidden from the graphic model. Deciding decision item D6 to false in Fig. 1 is an example for the latter case: the antecedent of constraint C1 becomes false and thus the consequent of this constraint does not need to be enforced anymore. This means that the constraint is satisfied and will be hidden from the graphic model. If we leave D6 undecided and decide D3 to true instead, we have an example for the first case, where constraint C1 needs to be minimized: the logical *and* operand in the consequent of the constraint is now satisfied and thus this clause needs to be removed from the constraint. This yields the following minimized form of C1: $D6 \Rightarrow D8 \text{ or } D9$.

In the ADORA tool we have solved and implemented the minimization problem with an existing algorithm. Out of several possible algorithms, we chose the Quine McCluskey algorithm [12] that is well known from computer hardware design for simplifying digital circuits. This algorithm is simple to implement and always finds a minimized form of the given constraint. The only disadvantage is that this algorithm computes only one minimized form of the constraint, even when there exist several ones. It could be that a different but equally minimal form of the constraint would be more intuitive to display in the graphical model. This, however, did not turn out as a considerable limitation yet, since it may occur only with very complex constraints.

After the minimizations of the graphical constraints have been computed, the ADORA tool weaves or removes all aspect containers and join relationships associated with the decision taken by the engineer as well as those aspect containers and join relationships associated with the decisions taken due to

constraint propagations. The weaving semantics builds on a slightly extended form of the weaving semantics introduced in [8], which focuses on modularizing cross-cutting concerns with aspects in ADORA. The details of the weaving semantics are beyond the scope of this paper. Examples are provided in the next section, see Fig. 4.

Finally, as Fig. 3 shows, the last task is re-calculating the constraint propagations as described above in subsection II B and updating the *ifTrue* and *ifFalse* columns of the decision table accordingly.

As a final result, ADORA displays a new partially (or fully) woven product line variability model which has less variability than before and is consistent with all constraints. The engineer can now continue with further variability binding decisions in the derivation process.

IV. EXAMPLE: SEMI-AUTOMATED STEPWISE AND INCREMENTAL PRODUCT DERIVATION WITH ADORA

In Fig. 4 we show an incremental, stepwise product derivation in five steps, as implemented in the ADORA tool. As an example, we use the automation devices product line that we introduced in section 2. Fig. 1 presents the fully unbound product line domain model which is the basis for this product derivation process. We assume that a group of engineers wants to derive the requirements specification for a concrete automation device from this product line.

Let's assume that the engineers want the product to be an intelligent field device. As they can see from the product line domain model (cf. Fig. 1), selecting this variant requires to dismiss the variant *Supervisory Unit* (*ifTrue* column of decision D2 and the graphical constraint VP1). The engineers choose to select this variant and set the decision value of the decision item D2 to true in the ADORA tool. As a consequence, the ADORA tool propagates the constraint in the *ifTrue* column of decision item D2 and sets decision item D1 to true. Then the tool hides the variation point constraint VP1 because it is now satisfied. Next, the tool performs the weaving operations associated with D2 and D1: The model fragment contained in the *Intelligent Field Device* aspect is woven into the *Automation Device* object set. The *Supervisory Unit* aspect and the join relationship labeled D1 are removed from the model because D1 has been set to false as a constraint propagation. Weaving the *Intelligent Field Device* aspect further requires a redirection of all its incoming join relationships (i.e. the join relationships of the sub-variants of this variant). These join relationships receive new target join points, which are now inside the *Automation Device* object set. Finally, ADORA re-calculates the constraint propagations for all remaining decision items. In this case there are no changes, except that the *ifTrue* and *ifFalse* values disappear for decision items D1 and D2 as they have been decided in this step. Diagram 1 in Fig. 4 shows the result of this first product derivation step. All concerned aspects and join relationships are highlighted. Removed items are marked with a cross. Note that these markups are *not* done by the ADORA tool, but have been added manually here for convenience of the reader.

In the second step, the engineers decide to choose the communication standard B to be part of their product and thus set decision item D4 to true. For this decision, no constraint propagations are necessary, because decision item D4 has no values in the *ifTrue* and *ifFalse* columns. However, D4 is involved in the variation point constraint VP2. This constraint (minCard 1, maxCard 3 on D3, D4, D5) is satisfied when D4 is decided to true and will be hidden, hence. Next the aspect *Communication Standard B* is woven into the *Automation Device* object set and into the *Configuration Tool* object, according to the join relationships associated with D4 in Diagram 1 in Fig. 4. Finally, the constraint propagations are re-calculated – again there are no changes. Diagram 2 in Fig. 4 shows the resulting intermediate model after step 2.

In step 3 the engineers decide *not* to choose the *Ring Buffer* variant for an implementation of the persistency component and set decision item D8 to false. This decision again does not trigger any constraint propagation. However, it is involved in two variability constraints which both need to be minimized. The variation point constraint VP3 still puts a restriction on the remaining two variants, which now is an alternative. The global constraint C1 is reduced to $D6 \Rightarrow D3 \text{ and } D9$. This means that if the *Web Interface* variant is chosen in a subsequent step, the *Communication Standard A* and the *File System* variants must also be chosen. After these minimizations, the join relationship annotated with D8 is removed from the graphical model and the related aspect is removed as well. Finally, the constraint propagations are re-calculated. This time, the values for the decision items D6, D7 and D9 are modified. D7 and D9 (the *File System* persistency and the *No Persistency* options) are now mutually exclusive alternatives and, as a consequence of constraint C1, D6 (the *Web Interface* variant) definitely requires both D3 (*Communication Standard A*) and D9 (*File System* persistency) to be selected. Conversely, the value in the *ifFalse* column of decision item D3 tells the engineers that if they would decide not to select *Communication Standard A* in a subsequent step, the *Web Interface* variant could not be chosen anymore.

In step 4 the engineers can already reason on the basis of this newer and simplified decision table and graphical model. They decide to choose the *Web Interface* variant to be part of the derived product and set D6 to true. This consequently triggers three constraint propagations: D3 and D9 are set to true because of constraint C1, and D7 is set to false as a transitive consequence of setting D9 to true. All variability constraints are now satisfied and thus hidden from the graphic model. Then the weaving operations are performed and the constraint propagations re-calculated. In the resulting model, only D5 is still undecided.

In step 5 the engineers recognize that only the *Communication Standard C* variant is left as an option in this nearly full configuration. As there are no more constraints, the engineers can freely choose or dismiss this option. Here they decide that they don't need *Communication Standard C* and set the decision item D5 to false. Consequently the tool removes this aspect and the corresponding join relationship from the

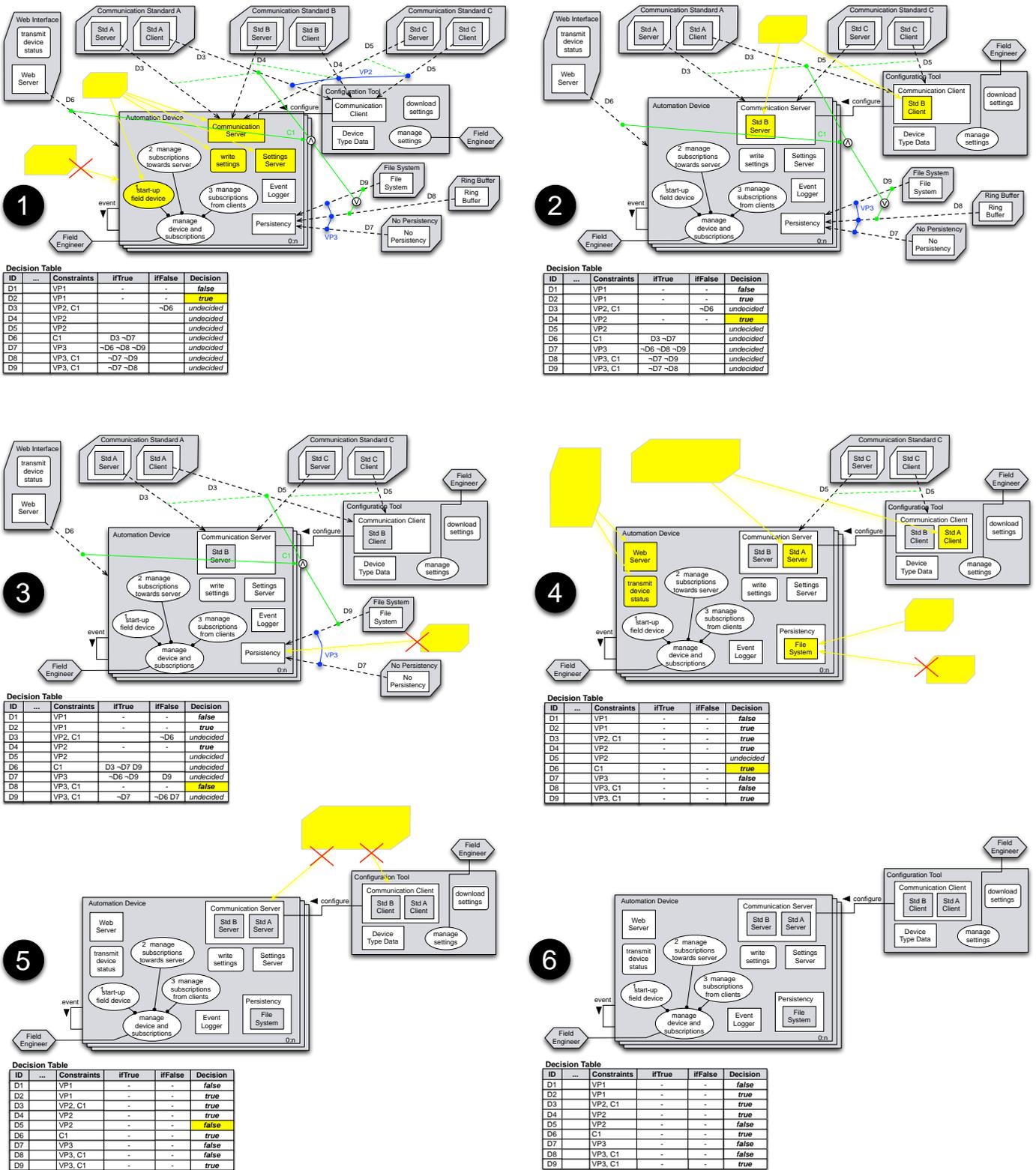


Fig. 4. An exemplary stepwise, incremental product derivation from the product line variability model of Fig. 1. Diagrams 1-5 show how the model changed after the respective derivation step. Diagram 6 shows the final product model. The yellow elements (light grey in B/W prints) and the crosses are only added for illustration purposes.

model. The result is a fully configured product requirements model which is correct (in terms of satisfying all constraints) by construction. Diagram 6 in Fig. 4 shows the final product model.

V. DISCUSSION AND CONCLUSION

A. Scalability

The ADORA language allows modeling the structure, behavior and user interaction of a software system in a single integrated hierarchical diagram. Using only one diagram for modeling a complete system requires sophisticated visualization techniques in order to scale, i.e. to keep large models comprehensible for humans users. In our previous work, we have developed such techniques, for example, fisheye zooming [13] [14], intelligent line-routing [15] and aspect-oriented modeling [8]. In [10] we have argued how these techniques can also be used for product line modeling in ADORA.

Concerning the runtime performance, our current implementation of partial weaving in the ADORA tool is not yet satisfactory for large models, while the performance of our SAT solving implementation doesn't seem to be a problem so far. We plan to do more detailed performance analyses and optimizations both for the our satisfiability solving solution and the aspect weaving in the ADORA tool.

B. Related Work

Stepwise and incremental product derivation is not entirely new. For example, Czarnecki et al. [5] [6] introduced a stepwise derivation process called 'specialization and multi-level configuration of feature models'. In comparison to our approach, this solution has two disadvantages. Firstly, the model elements describing a single feature are scattered over several diagrams (the approach relies on UML). Secondly, feature modeling allows only rather simplistic types of cross-tree constraints, as we argued in the introduction section.

Other authors have addressed the problem of automatic reasoning to leverage the complexity of variability modeling. Thüm et al. [16], for example, recently introduced a grounded framework for reasoning about edits to feature models. They define four particular types of edits that can be made in a feature model. One of them is specialization, which we also use for product derivation. Since they build on the foundations of Batory's work [11], their feature modeling solution allows arbitrarily complex constraints. However, their work so far focuses on feature modeling only. They do not provide any solution to integrate variability modeling with more detailed modeling of the functionality.

White et al. [17] presented a solution that creates mappings between variability models and equivalent constraint satisfaction problems (CSPs) and uses these CSPs for an automated calculation of a sequence of minimal feature adaptations between two different application feature models. As application feature models they typically consider an original feature configuration and a new one that has evolved during the development. White et al. focus on the evolution of an

already derived product configuration, while we focus on the product derivation itself in the first place.

Furthermore, there is a range of commercial and open source tools for feature modeling which also support product derivation. However, in terms of feature modeling and automated reasoning, these solutions are less advanced than the one of Thüm et al. [16] and they also have at least the same limitations as Czarnecki et al.'s feature modeling approach [6].

C. Conclusion

We have developed a new variability modeling approach that allows an inherently integrated modeling of features and requirements by building on aspect-oriented modeling, a new table-based boolean decision modeling solution and the ADORA language [7]. Our solution allows the description of arbitrarily complex variability constraints, implements an automated analysis and constraint propagation of these variability constraints and supports incremental and stepwise product derivation. This significantly reduces the complexity and the cognitive load for the model users and improves the understanding of the consequences and the impact of concrete variability binding decisions by human engineers involved in the product derivation process. We claim that already for intermediately complex product lines, such a stepwise and incremental approach to product derivation becomes necessary in order to enable an efficient and intuitive derivation of consistent and valid products.

In our current tool implementation, we have fully implemented the support for stepwise and incremental product derivation that we described in this paper. However, there is still room for improvements and extensions. For example, the generation of human-friendly graphical layouts with our weaving implementation is still a challenge. We did not yet find any really suitable algorithms that always generate satisfying layouts. Further, for our automated reasoning and constraints propagation solution, we do not yet calculate adequate intuitive and minimal sets of constraint propagations needed when already bound decisions are reverted. This is challenging because also the order of the previously taken decisions and their constraint propagations need to be taken into account. The runtime performance of our tool implementation, in particular when weaving models, also needs improvement. Finally, we plan to do empirical evaluations in the near future in order to validate the usefulness of our approach and the support that ADORA provides for real-world product line modeling problems.

ACKNOWLEDGMENT

We would like to thank Ivo Vigan for his programming work and his advice for implementing a SAT solver for ADORA.

REFERENCES

- [1] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep., November 1990.

- [2] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Generic semantics of feature diagrams," *Comput. Netw.*, vol. 51, no. 2, pp. 456–479, 2007.
- [3] M. Mannion, "Using first-order logic for product line model validation," in *SPLC'02: Proceedings of the Second Software Product Line Conference.*, ser. Lecture Notes in Computer Science, G. J. Chastek, Ed., vol. 2379. Springer, 2002, pp. 176–187.
- [4] S. Jarzabek, W. C. Ong, and H. Zhang, "Handling variant requirements in domain modeling," *J. Syst. Softw.*, vol. 68, no. 3, pp. 171–182, 2003.
- [5] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.
- [6] K. Czarnecki, M. Antkiewicz, C. H. P. Kim, S. Lau, and K. Pietroszek, "fmp and fmp2rsm: Eclipse plug-ins for modeling features using model templates," in *OOPSLA Companion*, 2005, pp. 200–201.
- [7] M. Glinz, S. Berner, and S. Joos, "Object-oriented modeling with ADORA," *Inf. Syst.*, vol. 27, no. 6, pp. 425–444, 2002.
- [8] S. Meier, T. Reinhard, R. Stoiber, and M. Glinz, "Modeling and evolving crosscutting concerns in ADORA," in *Aspect-Oriented Requirements Engineering and Architecture Design, 2007. Early Aspects at ICSE: Workshops in*, May 2007.
- [9] R. Stoiber and M. Glinz, "Modeling and managing tacit product line requirements knowledge," in *2nd International Workshop on Managing Requirements Engineering Knowledge (MaRK'09)*. IEEE CS, 2009.
- [10] R. Stoiber, T. Reinhard, and M. Glinz, "Visualization support for software product line modeling," in *Proceedings of SPLC'08 (Second Volume)*. *2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPL'08)*, 2008, pp. 313–322.
- [11] D. S. Batory, "Feature models, grammars, and propositional formulas," in *SPLC'05: Proceedings of the 9th International Software Product Line Conference.*, ser. Lecture Notes in Computer Science, J. H. Obbink and K. Pohl, Eds., vol. 3714. Springer, 2005, pp. 7–20.
- [12] E. J. McCluskey, "Minimization of boolean functions," *Bell System Technology Journal.*, vol. 35, no. 5, pp. 1417–1444, 1956.
- [13] C. Seybold, M. Glinz, S. Meier, and N. Merlo-Schett, "An effective layout adaptation technique for a graphical modeling tool," in *ICSE'03: Proceedings of the 25th International Conference on Software Engineering.*, May 2003, pp. 826–827.
- [14] T. Reinhard, S. Meier, R. Stoiber, C. Cramer, and M. Glinz, "Tool support for the navigation in graphical models," in *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 823–826.
- [15] T. Reinhard, C. Seybold, S. Meier, M. Glinz, and N. Merlo-Schett, "Human-friendly line routing for hierarchical diagrams," in *ASE '06: Proceedings of the 21st International Conference on Automated Software Engineering.*, Sept. 2006, pp. 273–276.
- [16] T. Thüm, D. Batory, and C. Kästner, "Reasoning about edits to feature models," in *ICSE '09: Proceedings of the 31st International Conference on Software Engineering.*, 2009, pp. 254–264.
- [17] J. White, D. Benavides, B. Dougherty, and D. C. Schmidt, "Automated reasoning for multi-step software product-line configuration problems," in *SPLC'09: Proceedings of the 13th International Software Product Line Conference*. IEEE CS, 2009.

Variability Modelling for Model-Driven Development of Software Product Lines*

Ina Schaefer

*Dept. of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
schaefer@chalmers.se*

Abstract—Model-driven development of software-intensive systems aims at designing systems by stepwise model refinement. In order to create software product lines by model-driven development, product variability has to be represented on every modelling level and preserved under model refinement. In this paper, we propose Δ -modelling as an generally applicable variability modelling concept that is orthogonal to model refinement. Products on each modelling level are represented by a core model and a set of Δ -models specifying changes to the core to incorporate product features. Core and Δ -models can be refined independently to obtain a more detailed model of the product line. Based on a formalization of Δ -modelling, we establish conditions that model refinement and model configuration commute resulting in an incremental model-driven development process.

Keywords-Software Product Lines; Variability Modelling; Model-driven Development; Model Refinement

I. INTRODUCTION

Model-driven development [1] of software-intensive systems aims at reducing design complexity by shifting the focus during system development from implementation to modelling. A model is an abstraction of a system with respect to certain system aspects. In model-driven development, an initial system model is successively refined by adding details relevant in particular design phases. A software product line [2], [3] is a set of systems with well-defined commonalities and variabilities. In order to use model-driven development in software product line engineering, the variability of the different products has to be represented within the used modelling concepts and preserved under model refinement.

The variability of products in software product lines is currently predominantly captured by feature models [4]. Features represent important product characteristics. A feature model determines a set of products by the set of valid feature configurations. However, features at the level of a feature model are merely labels [5]. Hence, feature-based variability has to be mapped to the modelling concepts used on each modelling level [6] in order to design a product line by a model-driven development process.

Existing approaches to integrate feature-based variability into modelling languages can be classified in two main directions [7]. First, negative variability-based

approaches consider one model for all products of a product line that is augmented with variant annotations determining which model elements are present in which products [8], [9], [10], [11]. Second, positive variability-based approaches [6], [7], [12], [13], [14] associate model fragments to features and compose them for a given feature configuration. However, most approaches only focus on modelling concepts used on one modelling level and do not consider how the variability representation can be preserved under model refinement.

In order to define a seamless model-driven development process for software product lines, we propose Δ -modelling, a general concept integrating variability modelling with model refinement. On each modelling level, product line variability is represented by a core model and a set of Δ -models. The core model represents a valid product of the product line. Δ -models specify changes of the core model, i.e., additions, modifications and removals of model fragments, in order to capture further products. An application condition attached to a Δ -model determines for which feature configuration a Δ -model is applicable. A product model for a feature configuration is obtained by applying the modifications specified by the Δ -models with valid application conditions.

For refinement, the core model and every Δ -model are transformed independently into a more detailed core model or Δ -model, respectively. The internal structure of the core and Δ -models, as well as the application conditions of the Δ -models are preserved. If the specified modifications in the refined Δ -models satisfy local refinement compatibility conditions, a refined product model for a feature configuration can be obtained in two ways: first, the product model is configured on the higher level of abstraction and afterwards transformed to a refined model; or second, the core and Δ -models are refined and afterwards configured by applying the modifications of the refined Δ -models to the refined core model. The commutativity of model refinement and model configuration builds the basis for incremental model-driven development of software product lines.

The Δ -modelling concept provides an integrated variability modelling approach for model-driven development of software product lines. Its main characteristics are:

- Δ -modelling is independent of a concrete modelling or implementation language. It can be instantiated to concrete modelling or implementation languages by defining the semantics of Δ -application.

*This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) and by the European project HATS, funded in the Seventh Framework Program.

- Combinations of features can be explicitly captured by flexible application conditions attached to Δ -models.
- Modular and evolutionary system development is facilitated by adding Δ -models to an existing model.
- Model refinement is orthogonal to variability modelling. Core and Δ -models are refined independently such that variability is expressed by the same structural concepts on all modelling levels.
- The commutativity of model configuration by Δ -application and model refinement provides the basis for an incremental development process by stepwise refinement of core and Δ -models.

The outline of this paper is as follows: In Section II, we review related work. In Section III, we explain the Δ -modelling approach at an example of a trading system product line. In Section IV, we formalize Δ -modelling and extend this formalization to model refinement in Section V. Section VI concludes with an outlook to future work.

II. RELATED WORK

Model-driven engineering for software product line development is proposed in [10], [15] in order to resolve product variability by model transformations. A model in the problem domain, usually a feature model, is transformed into a model in the solution domain, e.g., a product model [7], product architecture [16] or product implementation [17].

The existing approaches to represent feature-based variability can be classified into two main directions [7]. Annotative approaches specify negative variability. They consider one model representing all products of a product line. Variant annotations, e.g., using UML stereotypes [8], [9], [10], [11], define which parts of the model have to be removed to derive the model of a concrete product. [5] associates presence conditions to modelling elements to be removed in certain feature configurations.

Compositional approaches capture positive variability. Model fragments are associated with features and composed for a particular feature configuration. A prominent example is the AHEAD [12] approach. A product is built by stepwise refinement of a base module with a sequence of feature modules. In [6], [7], [13], models are constructed by aspect-oriented modelling techniques. [14] applies model superposition to compose model fragments. In [18], a product model is obtained by composition and refined by model transformation. [19] propose to represent model variability by a base model and associated variability and resolution models determining how modelling elements of the base model have to be replaced for a particular product model. The base model is similar to the core model in the Δ -modelling approach while variability and resolution models correspond to Δ -models, but are not directly connected product features.

Most of the above approaches only focus on the representation of variability on a single modelling layer. In [9], different modelling levels during system development are

considered, but variability resolution is based on textual decision models that are separated from the system models. In contrast, Δ -modelling facilitates a seamless representation of variability inbetween different modelling layers.

The notion of program deltas is introduced in [20] to describe the modification of an object-oriented program, e.g., by introduction of new fields or extension of methods. The mapping of collaborative features to models in [6] is similar to Δ -models. Collaborative features can modify a core model by additions, removals and modifications, but require a one-to-one relationship to a feature. In [21], Δ -modelling is presented as an approach to develop product line artifacts suitable for automated product derivation.

Feature-oriented model-driven development (FOMDD) [22] combines feature-oriented programming (FOP) with model-driven engineering. In FOMDD, a product can, first, be composed from a base module and a sequence of feature modules and afterwards transformed to another product model. Second, the base module and the feature modules can be transformed and then composed to a transformed product model. This is similar to the commutativity between model refinement and model configuration in Δ -modelling. FOP can be seen as a special case of Δ -modelling. Feature modules are always associated to exactly one feature, whereas Δ -models explicitly consider combinations of features. In feature modules, only additions and modifications can be specified. In contrast, Δ -models may contain removals of model parts. While the base module in FOP is fixed by the mandatory features, in Δ -modelling, any valid product can be chosen as core model enabling a flexible product line design.

III. VARIABILITY MODELLING USING Δ -MODELS

Figure 1 shows an overview of the model-based development process for software product lines using the Δ -modelling approach. First, an initial model of the product line is created that captures the variability of the feature model. From this initial model on a high level of abstraction, successively refined models are constructed that describe more detailed aspects of the considered products.

On each modelling level, product variability is captured by a core model and a set of Δ -models. A core model corresponds to a valid product of the product line. Δ -models specify changes to the core model by additions, modifications and removals of model fragments in order to represent further products. An application condition is attached to every Δ -model determining for which feature configurations the specified changes are to be carried out. In order to obtain a product model for a feature configuration, the changes specified by Δ -models with valid application condition are applied to the core. The concept of Δ -modelling to express variability is independent of a concrete modelling language. The modelling constructs used on each modelling level can be chosen to appropriately represent the considered system aspects. The application conditions attached to the Δ -models create

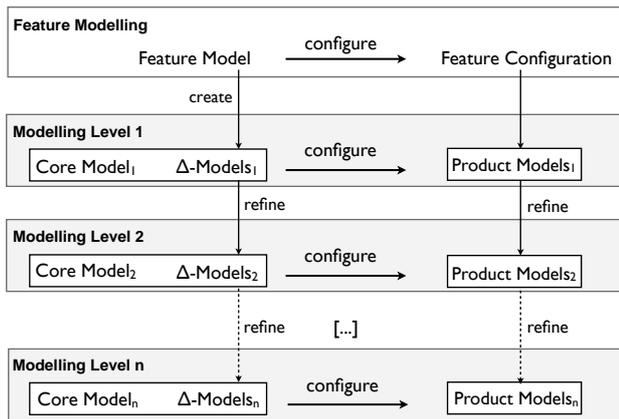


Figure 1. Model-driven development with Δ -Modelling

the connection between the features in the feature model and product variability on the different modelling levels. If the selection of a feature influences the choice of the modelling language, e.g., if a feature refers to the used implementation framework, core and the Δ -models can be seen tuples containing the specifications of the core and Δ -models in the respective modelling formalisms, while the general variability structure is preserved.

The step from a feature model of a product line to the initial core model and the set of Δ -models is a creative process, since product line variability can in general be represented in different ways. The variability structure provided by the initial modelling level provides the variability structure of the lower, more refined modelling levels (cf. Figure 1). A core model is refined to a more detailed core model. Δ -models are refined to more detailed Δ -models with the same application condition. An important property of the refinement between two modelling levels is that it commutes with model configuration by Δ -application. This means that a refined configured product model can be obtained in the following two ways. First, the product model for a feature configuration is configured from the core model and the applicable Δ -models and afterwards refined. Second, the core model and the Δ -models are refined, such that afterwards the refined product can be configured. This commutativity property provides the basis for an incremental model-based development process by stepwise model refinement.

Example We illustrate variability modelling based on Δ -models at the case example of a software product line of trading systems. The Common Component Modeling Example (CoCoME) [23] describes a software system handling payment transactions in supermarkets. It was extended to a software product line in [24]. The variability of the products are expressed in the feature diagram [4] shown in Figure 2. Mandatory features are represented by a filled circle, optional features with an empty circle. Alternative features are specified with a filled triangle if at least one feature has to be selected or by an empty triangle if exactly one features has to be selected. Constraints between features are represented by explicit links. A

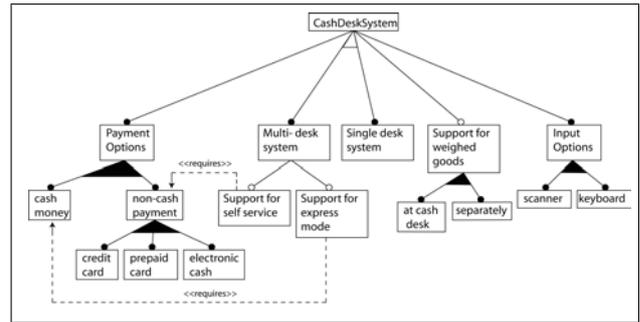


Figure 2. Feature Model for the CoCoME Software Product Line

product in the trading system product line has different payment options, i.e., cash payment or payment by credit card, prepaid card or electronic cash. At least one payment option has to be chosen for a valid configuration. Product information can be entered using a keyboard or a scanner, where at least one option has to be selected. Furthermore, the system has optional support to weigh goods, either at the cash desks or at separate facilities. A trading system can be configured as a single-desk system with only one cashier or as a multi-desk system with several of cashiers. A multi-desk system can optionally comprise an express mode which requires cash payment or a self-service mode requiring non-cash payment.

A core model represents a product for a valid feature configuration. Thus, it can be developed by well-established single application engineering techniques as a standard product model. In the example, the feature configuration containing the keyboard, the cash payment and the single-desk system features is selected as core configuration.

Component Modelling Level In our example, we start the model-based development process at the component level by representing the core model by a UML component diagram [25] and its variability by component diagram Δ -models that are an extension of UML component diagrams with annotations for the specified changes. In a second step, the component diagrams are refined to UML class diagrams showing in more detail how the components are implemented. Figure 3 depicts the component diagram specifying the core product of the trading system product line with the keyboard, cash payment and single-desk system features. It contains a Cash Desk component dealing with cash payment and an Inventory component keeping the store inventory. Every time a product is entered at the cash desk, the price of the product is requested from the inventory.

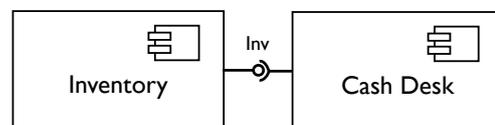


Figure 3. Core Component Diagram

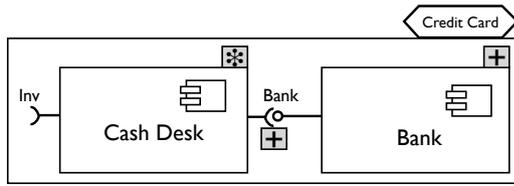


Figure 4. Component Diagram Δ-Model

Figure 4 depicts the component diagram Δ-model containing the modifications of core component diagram to include credit card functionality. A Bank component has to be added specified by the + annotation at the Bank component. Additionally, the Cash Desk component has to be modified to handle credit card payment which indicated by * annotation at the Cash Desk component. In order to realize the communication with the Bank component, an required interface and a corresponding connection to the bank component have to be added. The application condition of this Δ-model (in the top right hand corner) defines that the modifications are carried out if the credit card feature is selected. In Figure 5, the component diagram for a single-desk system containing keyboard input, cash payment and credit card payment is depicted that results from applying the Δ-model for credit card payment (cf. Figure 4) to the core component diagram (cf. Figure 3).

Class Modelling Level Each component of the trading system product line can be refined to a class diagram. The class diagram represents the internal component structure and can be used as basis for an implementation. The interactions between the components are not considered on the class diagram level because they are already captured on the component modelling level. In the Δ-modelling approach, core and Δ-models are refined independently. A refined product model is obtained by applying the refined Δ-models to the refined core model.

Figure 6 shows the class diagram for the Cash Desk component contained in the core. It comprises a Cash Desk class implementing the main functionality of the cash desk, a Keyboard class handling the input from the keyboard and a Display class providing output to a display. Figure 7 depicts the class diagram Δ-model specifying the modifications of the core class diagram Cash Desk component to incorporate the credit card feature. In order to provide credit card payment functionality, a Card Reader class is required. The Cash Desk class is modified by adding a reference to the bank, by adding methods to deal with the credit card payment and by modifying the existing payment methods.



Figure 5. Product Component Diagram

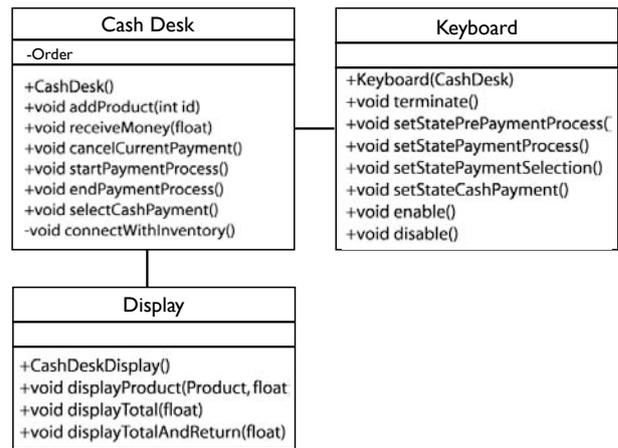


Figure 6. Core Class Diagram

Model Refinement and Configuration The class diagram for the configured Cash Desk component with the basic features and the credit card feature can be obtained in two different ways. First, the core model and the Δ-model on the component modelling level (cf. Figure 3 and 4) can be refined to a core and Δ-model on the class diagram level (cf. Figures 6 and 7) and configured to a class diagram by the standard configuration procedure. Second, a component diagram including the Cash Desk component with the basic features and the credit card feature can be configured on the component diagram level (cf. Figure 5). Afterwards, the configured Cash Desk component can be refined to a class diagram specifying the component's structure in more detail.

Model refinement and model configuration commute in the example because the refinement of the component diagram Δ-models is compatible with model configuration. Compatibility requires that for a component added (or removed) by a component diagram Δ-model, in the class diagram refinement of this Δ-model, all parts of the class diagram are specified as added (or removed) as well. If a component is modified in a component diagram Δ-model, in the refined class diagram, the parts of the class diagram

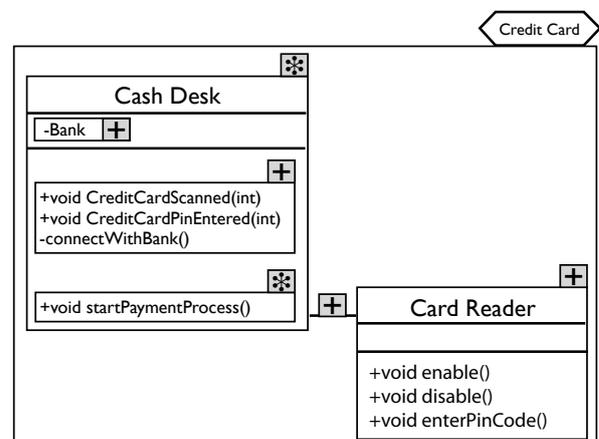


Figure 7. Class Diagram Δ-Model

may be specified as added, modified or removed. However, the change operations resulting from refinement of a modification operation have to satisfy a local refinement compatibility condition. This condition requires that the class diagram obtained by applying the refined component Δ -model to the class diagram of a refined core component is the same as the class diagram refinement of the same component configured on the component diagram level.

IV. FORMALIZING Δ -MODELLING

Variability modelling using Δ -models is a general approach that is not limited to specific modelling concepts, such as component or class diagrams used in Section III. Δ -modelling can be applied to any modelling or implementation language by defining the semantics of the change operations specified in the Δ -models for the concrete language. The number of modelling layers also depends on the concrete application and is not limited by the Δ -modelling approach. For instance, in Section III, use case diagrams separated into core and Δ -diagrams could be used to represent the requirements of the set of systems under development and subsequently be refined to component diagrams.

In order to show the general applicability of Δ -modelling, we base the following formalization on a general notion of models. A *model* contains a set of modelling elements E that can, for instance, represent components or classes (by their names). The set of modelling elements E describes the (domain-specific) concepts used in the models. Furthermore, a model contains relations between modelling elements representing correspondences, such as connections between required and provided interfaces in component diagrams. For simplicity, we restrict our notion of a model to contain only one binary relation R over the set of modelling elements E . This allows us to consider relations formally while keeping the model and its formal treatment simple. In a concrete model instantiation, a set of relations can be defined to express different relationships between elements.

Definition 1 (Models): Let E be a set of modelling elements. A *model* M is a tuple $M = (E, R)$ where $R \subseteq E \times E$ is a relation over the modelling elements.

A *core model* represents a product for a valid feature configuration. This allows treating the core model in the same way as any product model. We define the set of valid feature configurations as a subset of the powerset of the set of features.

Definition 2 (Core and Product Models): For a set of features $F = \{f_1, \dots, f_n\}$, let $\mathcal{F} \subseteq \mathcal{P}(F)$ denote the set of valid feature configurations. A core model (product model) is a triple $C = (E, R, f)$ where $f \in \mathcal{F}$ is a valid feature configuration, and (E, R) is a model representing the feature configuration f .

A Δ -model specifies changes to a core model to model other products. For a core model $C = (E, R, f)$, a Δ -model defines additions, modifications and removals of modelling elements $e \in E$, and additions and removals of

tuples in the relation R . A Δ -model has an *application condition* determining under which feature configurations the specified changes have to be carried out. Application conditions are logical (e.g., Boolean) constraints over the features contained in the feature model. A Δ -model does not necessarily refer to exactly one feature, but potentially to a combination of features. This allows very flexible Δ -models as combinations of features can be handled individually. For example, if a feature model contains two features A and B , the Boolean constraint $(A \wedge \neg B)$ denotes that the modifications are only carried out for a feature configuration if feature A is selected and feature B is not selected. The granularity of the application conditions determines the number of Δ -models that have to be created to ensure that all features present in the feature model are appropriately captured.

Definition 3 (Δ -Models): A Δ -model over a model $M = (E, R)$ is a tuple $\Delta = (\varphi, Op)$ where the application condition φ is a constraint over the set of features $F = \{f_1, \dots, f_n\}$ and $Op = \{op_1, \dots, op_m\}$ is a set of modification operations over the model M with

$$op_i ::= \text{add } e \mid \text{mod } e \mid \text{rem } e \mid \text{add } r(e_1, e_2) \mid \text{rem } r(e_1, e_2)$$

In order to obtain a product model for a feature configuration $f \in \mathcal{F}$, all Δ -models with valid application condition for the feature configuration f are applied to the core model. This can involve different Δ -models that are applicable for the same feature. To limit the occurrence of conflicts between changes targeting the same modelling elements and relations, first all additions, then all modifications and finally all removals are performed. In order to express this ordering formally, we assume that Δ -models are *normalized*, i.e., their change operations contain only additions, only modifications, or only removals. A Δ -model $\Delta = (\varphi, Op)$ can be normalized by splitting it into three disjoint normalized Δ -models $\Delta_a = (\varphi, Op_a)$, $\Delta_m = (\varphi, Op_m)$ and $\Delta_r = (\varphi, Op_r)$ such that $Op = Op_a \uplus Op_m \uplus Op_r$. We call a set of Δ -models $\Delta = \{\Delta_1, \dots, \Delta_n\}$ *sorted* if and only if there exist i, j with $1 \leq i \leq j \leq n$, such that $\Delta_1, \dots, \Delta_i$ contain only additions, $\Delta_{i+1}, \dots, \Delta_j$ contain only modifications, and $\Delta_{j+1}, \dots, \Delta_n$ contain only removals. The operation $\nu(\Delta)$ transforms a set of Δ -models into a set of normalized and sorted Δ -models. The application function $apply(M, Op)$ modifying a model M by the change operations Op is defined in Definition 5.

Definition 4 (Configuration): Let $C = (E, R, f_c)$ be a core model and $\Delta = \{\Delta_1, \dots, \Delta_n\}$ be a sequence of normalized and sorted Δ -models with $\Delta_i = (\varphi_i, Op_i)$ for all i . For a feature configuration $f \in \mathcal{F}$, a product model $P_f = (E_P, R_P, f)$ is configured by $conf((E, R), \{\Delta_1, \dots, \Delta_n\}, f)$ where for a model $M = (E_M, R_M)$, its configuration is defined by $conf(M, \{\Delta_1, \dots, \Delta_n\}, f) =$

$$\begin{aligned} & conf(apply(M, Op_1), \{\Delta_2, \dots, \Delta_n\}, f) && \text{if } f \models \varphi_1 \\ & conf(M, \{\Delta_2, \dots, \Delta_n\}, f) && \text{otherwise} \end{aligned}$$

and $conf(M, \emptyset, f) = (E_M, R_M, f)$.

The ordering in which the change operations specified in a single Δ -model are applied to a model is not fixed, which can also be seen as simultaneous application of the specified changes. The same modelling element can be added several times, but only occurs once in the model. Similarly, a tuple in the relation is added only once, if the related modelling elements are contained in the model. The modification of a modelling element also causes that the element is replaced in all relational tuples in which the original modelling element is contained. If an element is removed, also all relational tuples containing this element are removed from the relation. The application of a change operation is undefined if a modelling element e is modified that is not contained in the core or added before by another Δ -model, or if a modelling element or a relational tuple is removed, that is not contained in the core or added before by another Δ -model.

Definition 5 (Δ -Application): The application of a set of change operations $Op = \{op_1, \dots, op_n\}$ to a model $M = (E, R)$ is defined by the application function *apply*:

- $apply(M, \emptyset) = M$
- $apply(M, Op) = apply(apply(M, op_i), Op \setminus \{op_i\})$
- $apply(M, \text{add } e) = (E \cup \{e\}, R)$
- $apply(M, \text{mod } e) = (E \setminus \{e\} \cup \{e'\}, R')$, if $e \in E$ where $e' \in E$ is the result of the modification of $e \in E$ and $R' = \{(e_1, e_2) \mid (e_1, e_2) \in R, e_1, e_2 \neq e\} \cup \{(e', e_2) \mid (e, e_2) \in R, e_2 \neq e\} \cup \{(e_2, e') \mid (e_2, e) \in R, e_2 \neq e\} \cup \{(e', e') \mid (e, e) \in R\}$
- $apply(M, \text{rem } e) = (E \setminus \{e\}, R')$, if $e \in E$ where $R' = \{(e_1, e_2) \mid (e_1, e_2) \in R \wedge e_1, e_2 \neq e\}$
- $apply(M, \text{add } r(e_1, e_2)) = (E, R \cup \{r(e_1, e_2)\})$, if $e_1, e_2 \in E$
- $apply(M, \text{rem } r(e_1, e_2)) = (E, R \setminus \{r(e_1, e_2)\})$, if $e_1, e_2 \in E$ and $r(e_1, e_2) \in R$.

Despite using normalized and sorted Δ -models during configuration, there can still be conflicts between the change operations specified in different Δ -models. A conflict occurs if a modelling element or tuple in a relation is added and removed by two different Δ -models, if a modelling element is modified and removed by two different Δ -models or if a modelling element is modified by two different Δ -models. This indicates that the granularity of the Δ -models and their application conditions is too coarse. Conflicts can be removed by splitting Δ -models and refining them to explicitly cover the conflicting feature combinations.

Definition 6 (Conflicts in Δ -Models): A set of Δ -models $\Delta = \{\Delta_1, \dots, \Delta_n\}$ contains a conflict if for a feature configuration $f \in \mathcal{F}$, there are Δ -models Δ_i and Δ_j with $i \neq j$, $f \models \varphi_i$ and $f \models \varphi_j$, and there exists $e \in E$, or $e_1, e_2 \in E$ and $r(e_1, e_2) \in R$, such that one of the following holds:

- $\text{add } e \in Op_i$ and $\text{rem } e \in Op_j$
- $\text{add } r(e_1, e_2) \in Op_i$ and $\text{rem } r(e_1, e_2) \in Op_j$
- $\text{mod } e \in Op_i$ and $\text{rem } e \in Op_j$
- $\text{mod } e \in Op_i$ and $\text{mod } e \in Op_j$

A core model $C = (E, R, f_b)$ and a set of Δ -models Δ are *well-defined* if for all valid feature configurations $f \in \mathcal{F}$, all applications of Δ -operations are defined and there are no conflicts between any two Δ -models. Well-definedness is a prerequisite for commutativity of model configuration and model refinement.

V. MODEL REFINEMENT AND CONFIGURATION

Based on the formalization of the Δ -modelling approach in Section IV, model refinement of core and Δ -models can be defined. A model is transformed to a more detailed model by refining the contained modelling elements to models themselves, as in the example in Section III, components are refined to class diagrams showing their internal structure. Relations between modelling elements are not considered for refinement, such as connections between components are only relevant on the component modelling level.

Definition 7 (Model Refinement): The refinement operation *refine* maps every modelling element $e \in E$ to a model M such that $refine(e) = M_e = (E_e, R_e)$. The refinement M' of a model $M = (E, R)$ is defined by $refine(M) = (\{M'' \mid M'' = refine(e), e \in E\}, \{r(refine(e_1), refine(e_2)) \mid r(e_1, e_2) \in R\})$

The core model and all other product models can be refined using the above definition of model refinement. In order to refine Δ -models, the specified addition, modification and removal operations on modelling elements and relations have to be refined. The addition of a modelling element is refined to a set of addition operations for the elements and the relational tuples of the model obtained by refining the modelling element. The removal of a modelling element is refined to a set of remove operations for the modelling elements and relational tuples of the model resulting from refining the modelling element. The modification of a modelling element is refined to a set of addition, modification and removal operations for modelling elements and relational tuples obtained by refining the modelling element. Change operations for relations are removed during Δ -refinement. The application condition of a Δ -model remains unchanged such that the variability structure is preserved. In the example in Section III, the component diagram Δ -model (cf. Figure 4) is refined to a class diagram Δ -model (cf. Figure 7) according to the following definition.

Definition 8 (Δ -Refinement): The refinement of a Δ -model $\Delta = (\varphi, Op)$ is defined by $refine(\Delta) = (\varphi, refine(Op))$ where $refine(\{op_1, \dots, op_n\}) = \{refine(op_1), \dots, refine(op_n)\}$ and

- $refine(\text{add } e) = \{\text{add } e' \mid e' \in E_e\} \cup \{\text{add } r(e'_1, e'_2) \mid r(e'_1, e'_2) \in R_e, e'_1, e'_2 \in E_e\}$
- $refine(\text{rem } e) = \{\text{rem } e' \mid e' \in E_e\} \cup \{\text{rem } r(e'_1, e'_2) \mid r(e'_1, e'_2) \in R_e, e'_1, e'_2 \in E_e\}$
- $refine(\text{mod } e) = \{\text{op } e' \mid e' \in E_e\} \cup \{\text{op } r(e'_1, e'_2) \mid r(e'_1, e'_2) \in R_e, e'_1, e'_2 \in E_e\}$ where $\text{op} \in \{\text{add}, \text{rem}, \text{mod}\}$
- $refine(\text{add } r(e_1, e_2)) = refine(\text{rem } r(e_1, e_2)) = \emptyset$

The configuration of a refined model for a feature configuration $f \in \mathcal{F}$ is performed in three steps. First, the original model on the higher abstraction level is configured subject to the feature configuration f . Second, for every modelling element e included in the resulting product model, the refined core model restricted to the modelling element e is configured using the refined Δ -models restricted to the modelling element e subject to the feature configuration f . If a modelling element e is not contained in the core model, but introduced by a Δ -model, the refined core model restricted to the modelling element e is an empty model. Third, the refined modelling elements replace their non-refined version in the configured original model. The result is a model that contains models as modelling elements and relations between these models. In the example in Section III, a configured, refined model is a component diagram in which the components contain class diagrams showing their detailed internal structure. The restriction of a core model $C = (E, R, f_b)$ to a modelling element $e \in E$ is defined by $C|_e = (\{e\}, \emptyset)$ if $e \in E$ and by $C|_e = (\emptyset, \emptyset)$, otherwise. Further, we define $\Delta|_e = \{\Delta_1|_e, \dots, \Delta_n|_e\}$ as the set of Δ -models only modifying element $e \in E$ where $\Delta_i|_e = (\varphi_i, \{\text{op } e \in \text{Op}_i\})$ for $\text{op} \in \{\text{add}, \text{rem}, \text{mod}\}$.

Definition 9 (Configuration of Refined Models): Let $C = (E, R, f_c)$ be a core model, $\Delta = \{\Delta_1, \dots, \Delta_n\}$ a set of normalized and sorted Δ -models and $f \in \mathcal{F}$ a feature configuration. Let $P_f = (E_P, R_P, f) = \text{conf}((E, R), \{\Delta_1, \dots, \Delta_n\}, f)$ be the configured original model for the feature configuration f . Further, let $M_e = \text{conf}(\text{refine}(C|_e), \nu(\text{refine}(\Delta|_e)), f)$ be the refined configured modelling element for $e \in E_P$.

The refined configured model is defined by $P_r = \text{conf}_{\text{ref}}(\text{refine}(E, R), \text{refine}(\Delta), f)$ with $P_r =$

$$(\{M_e \mid e \in E_P\}, \{r(M_{e_1}, M_{e_2}) \mid r(e_1, e_2) \in R_P\}, f)$$

The commutativity of model refinement and model configuration by Δ -application constitutes the basis for the incremental model-based development of software product lines by stepwise refinement of core and Δ -models. The requirement for commutativity is that the refinement of the change operations specified in Δ -models is compatible with model configuration. For addition and removal operations, compatibility is ensured by the definition of Δ -model refinement in Definition 8. For the refinement of the modification operations, a local refinement compatibility condition has to be established. This local refinement compatibility condition requires that the result of applying the refined modification operation to the refined modelling element in core model is the same as the refinement of the modelling element that has been configured on the non-refined modelling level.

Definition 10 (Refinement Compatibility): Let $e' \in E$ be the result of applying the modification operation $\text{mod } e$ to the modelling element $e \in E$. The local refinement compatibility constraint for the operation $\text{mod } e$ holds iff

$$\text{apply}(\text{refine}(e), \text{refine}(\text{mod } e)) = \text{refine}(e')$$

The following theorem states that model refinement and model configuration commute if all modification operations satisfy refinement compatibility.

Theorem 1 (Commutativity): For a feature configuration $f \in \mathcal{F}$, a core model $C = (E, R, f_c)$ and a set of well-defined Δ -models $\Delta = \{\Delta_1, \dots, \Delta_n\}$ and a refinement refine on the modelling elements $e \in E$, if all modification operations $\text{mod } e \in \text{Op}_i$ satisfy the refinement compatibility condition, then it holds that $\text{refine}(\text{conf}((E, R), \Delta, f)) = \text{conf}_{\text{ref}}(\text{refine}(E, R), \text{refine}(\Delta), f)$.

Proof: By induction on the set of Δ -models and a case distinction on their add , mod , rem operations. For add and rem operations, the definition of Δ -refinement in Definition 8 is used. For mod operations, the refinement compatibility condition from Definition 10 is assumed. ■

Commutativity of model refinement and model configuration provides that basis for an incremental model-driven development process for software product lines. After the initial core and Δ -models have been created to capture the variability of the feature model, this initial variability structure is preserved on each modelling level by the independent refinement of core and Δ -models.

VI. CONCLUSION

Δ -modelling is a variability modelling approach for model-driven development of software product lines. Product variability is expressed by core and Δ -models on all modelling levels that are preserved under model refinement. For a concrete development process, the semantics of Δ -application has to be defined for the language concepts used on each modelling level. Model configuration by Δ -application can be automated, e.g., by aspect-oriented model weaving techniques [6], [7], [13] or model superimposition [14]. In [21], the Δ -modelling approach has been implemented using frame technology.

For future work, we aim at providing tool support and guidelines how to develop initial core and Δ -models for a given feature model. This will be complemented by modular analyses establishing conflict-freedom and well-definedness of core and Δ -models using existing variability analysis techniques based on confluence analysis. Besides, we want to extend Δ -modelling with explicit conflict resolution by imposing a partial order between Δ -models in order to avoid the normalization of Δ -models during configuration.

Acknowledgments The author wants to thank Arnd Poetzsch-Heffter and Alexander Worret for initial collaboration on the subject of this work.

REFERENCES

- [1] B. Selic, "The Pragmatics of Model-driven Development," *IEEE Software*, Sept 2003.
- [2] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2001.

- [3] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, Heidelberg, 2005.
- [4] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Carnegie Mellon Software Engineering Institute, Tech. Rep., 1990.
- [5] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," in *GPCE*, 2005.
- [6] F. Heidenreich and C. Wende, "Bridging the Gap Between Features and Models," in *Aspect-Oriented Product Line Engineering (AOPL'07)*, 2007.
- [7] M. Völter and I. Groher, "Product Line Implementation using Aspect-Oriented and Model-Driven Software Development," in *SPLC*, 2007, pp. 233–242.
- [8] T. Ziadi, L. Hérouët, and J.-M. Jézéquel, "Towards a UML Profile for Software Product Lines," in *Workshop on Product Family Engineering (PFE)*, 2003, pp. 129–139.
- [9] C. Atkinson, J. Bayer, , and D. Muthig, "Component-Based Product Line Development: The Kobra Approach," in *SPLC*, 2000.
- [10] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, and A. Solberg, "An MDA-based framework for model-driven product derivation," in *Software Engineering and Applications (SEA)*, 2004.
- [11] H. Gomaa, *Designing Software Product Lines with UML*. Addison Wesley, 2004.
- [12] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE Trans. Software Eng.*, vol. 30, no. 6, 2004.
- [13] N. Noda and T. Kishi, "Aspect-Oriented Modeling for Variability Management," in *SPLC*, 2008.
- [14] S. T. Sven Apel, Florian Janda and C. Kästner, "Model Superimposition in Software Product Lines," in *International Conference on Model Transformation (ICMT)*, 2009.
- [15] S. Deelstra, M. Sinnema, J. van Gurp, and J. Bosch, "Model Driven Architecture as Approach to Manage Variability in Software Product Families," in *Workshop on Model Driven Architecture: Foundations and Applications (MDAFA 2003)*, 2003.
- [16] G. Botterweck, L. O'Brien, and S. Thiel, "Model-driven Derivation of Product Architectures," in *Automated Software Engineering (ASE)*, 2007, pp. 469–472.
- [17] G. Botterweck, K. Lee, and S. Thiel, "Automating Product Derivation in Software Product Line Engineering," in *Software Engineering*, 2009, pp. 177–182.
- [18] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jézéquel, "Reconciling Automation and Flexibility in Product Derivation," in *SPLC*, 2008.
- [19] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen, "Adding Standardized Variability to Domain Specific Languages," in *SPLC*, 2008.
- [20] R. Lopez-Herrejon, D. Batory, and W. Cook, "Evaluating Support for Features in Advanced Modularization Technologies," in *ECOOP*, 2005.
- [21] I. Schaefer, A. Worret, and A. Poetzsch-Heffter, "A Model-Based Framework for Automated Product Derivation," in *Model-driven Approaches in Software Product Line Engineering (MAPLE 2009)*, 2009.
- [22] S. Trujillo, D. Batory, and O. Díaz, "Feature Oriented Model Driven Development: A Case Study for Portlets," in *ICSE*, 2007.
- [23] S. Herold *et al.*, "CoCoME - The Common Component Modeling Example," in *Common Component Modeling Example*, A. Rausch *et al.*, Eds. Springer-Verlag, 2008, pp. 16 – 53.
- [24] A. Worret, "Automated Product Derivation for the CoCoME Software Product Line: From Feature Models to CoBoxes," Master's thesis, University of Kaiserslautern, March 2009.
- [25] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology, 2004.

Using Incremental Consistency Management for Conformance Checking in Feature-Oriented Model-Driven Engineering

Roberto E. Lopez-Herrejon
Alexander Egyed

Institute for Systems Engineering and Automation
Johannes Kepler University Linz, Austria
{roberto.lopez, alexander.egyed}@jku.at

Salvador Trujillo
Josune de Sosa

IKERLAN Research Laboratory
Mondragon, Spain
{STrujillo,jdesosa}@ikerlan.es

Maidier Azanza

University of the Basque Country
San Sebastian, Spain
maider.azanza@ehu.es

Abstract—Feature-Oriented Model-Driven Engineering (FOMDE) is an approach that lies at the intersection of two complementary paradigms for software construction, Model Driven Engineering (MDE) and Software Product Line Engineering (SPLE). MDE aims at raising the abstraction level of application specification and automating the realization of these abstractions down to the platform level, while SPLE focuses on the synthesis of applications using a pre-planned set of assets. In Feature-Oriented, features are modules that contain all assets needed for their realization. The products of a Software Product Line (SPL) are synthesized by composing different combinations of features. When constructed following MDE, features also contain metamodels, models and model transformations. In this context, it is crucial to check that models, metamodels, and their compositions conform to (i.e. meet all the constraints of) their metamodels and meta-metamodels. In this problem statement paper we describe how to use incremental consistency checking to check this conformance. We sketch some of the potential benefits of this approach and highlight the open questions our work raised.

Index Terms—incremental consistency management; Model-Driven Engineering Software Product Lines; Feature Orientation; conformance checking

I. INTRODUCTION

Feature-Oriented Model-Driven Engineering (FOMDE) is an approach that lies at the intersection of two complementary paradigms for software construction, *Model Driven Engineering (MDE)* [1] and *Software Product Line Engineering (SPLE)* [2]–[4]. MDE aims at raising the abstraction level of application specification and automating the realization of these abstractions down to the platform level with a set of model to model and model to text transformations. On the other hand, SPLE focuses on the synthesis of applications using a pre-planned set of assets.

Feature-Oriented is a specific approach for constructing *Software Product Lines (SPL)* [5], [6]. In this approach, *features* are increments in program functionality [7]. Features are implemented in modules that contain all assets, that is, artifacts that they require for their realization. For example, a feature may be implemented with UML diagrams, scripts, XML files, configuration files, etc. The members of the SPL

are synthesized by composing different combinations of features. Tools that implement this approach provide mechanisms to compose these distinct artifact types in a uniform way. For the combination of Feature-Oriented and MDE, the features typically contain multiple models, metamodels and transformations [8].

In this context, it is crucial to check that models, metamodels, and their compositions conform to (i.e. meet all the constraints of) their metamodels and meta-metamodels respectively. The driving motivation of our problem statement paper is describing an approach to check this conformance.

Consistency checking derives from work on *Multi-View Modeling (MVM)* [9]. MVM advocates that multiple, different and yet related models are required to represent the perspectives and information needs of diverse system stakeholders throughout the development process [10], [11]. The elements in these distinct views have semantic relationships that must be expressed and maintained. Consistency rules capture and serve to enforce these semantic relationships. An example of MVM is UML where the different types of diagrams can represent the distinct views of a system [12]. A classical example of consistency rule in UML, between sequence and class diagrams, is that if a sequence diagram has a message m whose target is an object of class C , the class diagram of class C must contain method m .

Our work raises consistency checking beyond the traditional MVM perspective. We show how incremental consistency checking, a special form of consistency checking, can be used to check conformance as described above. The key is treating both model and metamodel composition similarly and generating conformance rules based on the well-formedness rules defined at the meta-metamodel. We sketch some of the potential benefits of this approach and highlight the open questions our work raised.

II. BACKGROUND

Our work brings together two, until now, disjoint research areas. In this section we present the basic background of both.

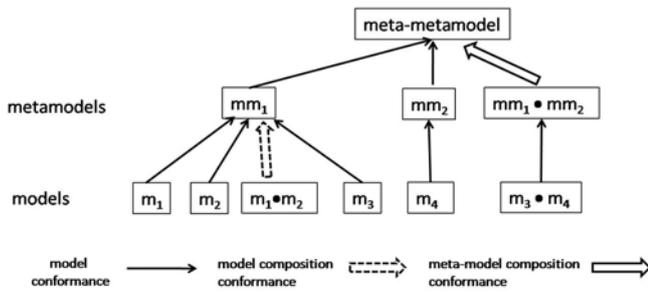


Fig. 1. Scenarios where model conformance is used

A. Feature-Oriented Model-Driven Engineering

Feature Oriented Model Driven Engineering (FOMDE) is a blend of *Feature Oriented Programming (FOP)* [6] and MDE that shows how products in an SPL can be synthesized in an MDE way by composing features to create models, and then transforming these models into executables [8]. FOP and MDE are complementary paradigms [13] and several case studies show the advantages of combining them [14], [15]. However, in these cases, features were implemented using XML and they were composed using XAK [16], that is, feature composition was purely text-based. This work laid out the foundations for our current research on FOMDE. The lack of conformance checking in FOMDE captured our interest and motivated this research.

Recent work on *Domain Specific Languages (DSL)* has raised the need of reusing metamodels as features of a SPL [17], [18]. The selection of different features produces thus different DSLs that are tailored to particular application scenarios. For this scenario to work properly, it is crucial to check that the composition of metamodels conforms to the meta-metamodel used and their corresponding models are kept conformant when their metamodels change.

In summary, in the context of FOMDE there are three main scenarios where checking conformance is important: *i)* model conformance to metamodel, *ii)* conformance during model composition, and *iii)* conformance during metamodel composition. These three scenarios are depicted in Figure 1. In this figure, m stands for model and mm for metamodel, and the dot indicates composition.

B. Incremental Consistency Checking

There exists an extensive body of work in consistency checking. Recent literature surveys identified over 30 approaches which rely on different formalisms to represent and validate consistency [19], [20]. They typically have in common that consistency is expressed via rules. A recent trend in consistency checking is work on incremental approaches which react to changes and evaluate only those rules on those model elements that are affected and can potentially cause an inconsistency. An advantage of these approaches is reduced verification time over systems that follow a batch strategy. A leading tool among the incremental approaches is UML/Analyzer [21], [22]. In this tool, when a model change

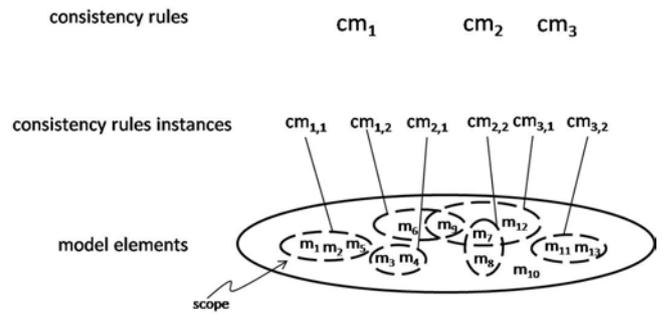


Fig. 2. Pictorial view of Incremental Consistency Checking

occurs, it automatically, correctly and efficiently identifies what consistency rules to evaluate and on what model elements. If inconsistencies are detected, they are highlighted for the user to take an appropriate corrective action.

Incremental consistency in UML/Analyzer works as follows. First the tool loads the model to analyze. Then it identifies the places where each consistency rule defined can be applied. A *consistency rule instance* is an application of a consistency rule, and its scope is the set of model elements that are part of the instance.

It is common that a model element is part of the scopes of multiple and distinct consistency rules instances. An example of this scenario is shown in Figure 2. This figure shows three consistency rules cm_1 , cm_2 , and cm_3 . For notational convention we denote these constraint rules with a suffix m , that stands for metamodel because these rules are defined in terms of metamodel elements, and a number subscript. We use a second subscript to denote instances of the constraint rules. Figure 2 shows instances of these rules such as $cm_{1,1}$ and $cm_{2,1}$. In this figure, model elements m_7 and m_9 belong to two distinct scopes.

The work of UML/Analyzer has been mostly used in the context of UML models; however, its underlying principles are applicable to any types of models and constraints [21], [22]. In the next section we show how these principles are adapted for checking model and metamodel conformance in the context of FOMDE.

III. CONFORMANCE CHECKING

In this section we draw a connection between incremental consistency checking and conformance checking for the three scenarios described above and sketch the potential benefits of this connection.

A. Conformance of Model to Metamodel

Let us illustrate the key ideas of UML/Analyzer with an example. Consider a hypothetical SPL of questionnaires. A typical questionnaire has a title and a brief introduction. The questions are grouped in blocks that have a header and a description. A block needs to have at least one question, and for each question requires two to four answer options. Figure 3 presents the metamodel for the features of this product line as an Ecore metamodel (a subset of UML class diagram).

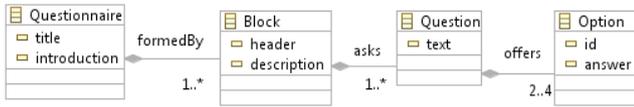


Fig. 3. Questionnaire metamodel

In FOMDD a feature contains models that are instances of a metamodel. Feature F, depicted in Figure 4, is an example in our questionnaire product line. For sake of simplicity we use an abbreviated object model to denote instantiation of the metamodel and annotate the relations amongst the objects using the aggregation names of the metamodel (*formedBy*, *asks*, and *offers*). This feature contains a block B1 with two questions (A and B), each with two answer options.

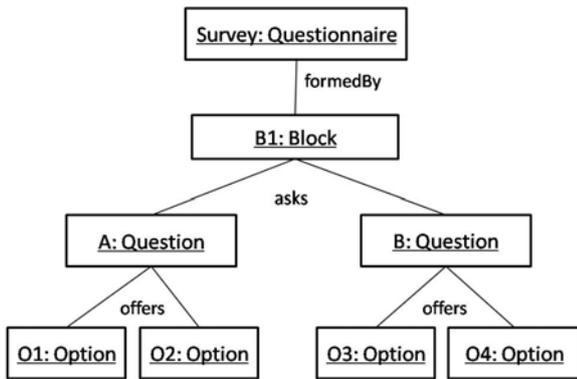


Fig. 4. Feature F, model instance of Questionnaire metamodel

The key for leveraging incremental consistency checking is using conformance rules as the consistency rules to check against. In other words, conformance rules and consistency rules have in common that they evaluate a portion of a model and return a boolean result, true if the rule holds or false otherwise. Consistency checkers can thus be used readily to check conformance rules. In our *Questionnaire* one of such conformance rules can be defined as follows ¹:

Conformance rule for aggregation. Let A and B be two classes. If an aggregation between A and B exists, an object of type A can aggregate n objects of type B where $lower \leq n \leq upper$.



Now we describe how to check conformance of the model in Figure 4 against this rule. In this figure, there are four instances of the conformance aggregation rule. The first instance contains *Question A* and its two *Option* answers from *offers* aggregation. Similarly, the second instance

¹To be more precise, the filled rhomb in this association denotes containment such that an object of type B can only be associated with one object of type A. For simplicity, we do not utilize this part of the standard semantics of this symbol as it is not relevant for our current exposition.

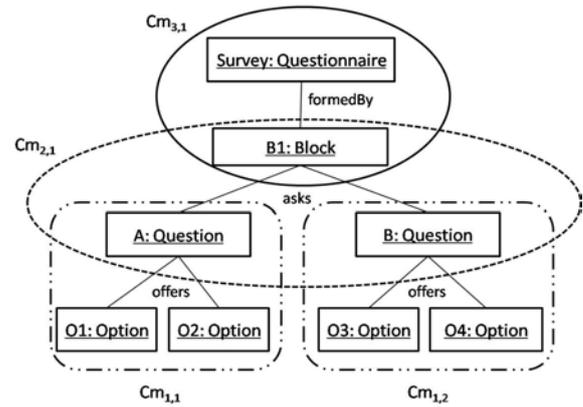


Fig. 5. Feature F with scopes

contains *Question B* and its two answers *Option*. The third instance contains a *Block* and the two questions, from *asks* aggregation. Finally, the fourth instance contains a *Questionnaire* object and one *Block*, from *formedBy* aggregation. In this model, the number of objects aggregated falls within the lower and upper limits of the corresponding rule instances, thus it conforms to its metamodel, according to this rule.

Notice however that this general conformance rule can be fine-tuned according to the types of the classes participating in the aggregation and their cardinality. We call this adaptation process *constraint rule generation*. For our *Questionnaire* metamodel the generated conformance rules are:

- cm_1 = Aggregation rule with A is *Question* and B is *Option*
- cm_2 = Aggregation rule with A is *Block* and B is *Question*
- cm_3 = Aggregation rule with A is *Questionnaire* and B is *Block*

The four instances of these generated rules are depicted in Figure 5. Thus, every feature will have an associated set of conformance rule instances with their corresponding scopes. In this figure the scopes are ²:

- $cm_{1,1} = \{A, O1, O2\}$
- $cm_{1,2} = \{B, O3, O4\}$
- $cm_{2,1} = \{B1, A, B\}$
- $cm_{3,1} = \{Survey, B1\}$

Our conformance rule for aggregation is a simplified example of the well-formedness rules that are commonly defined for meta-metamodels such as *Ecore* [23] and *MOF* [24]. Therefore, to check conformance of a model to a metamodel it is required to define all well-formedness rules from which conformance rules can be generated for a particular metamodel.

In addition to the well-formedness rules, it is possible to include constraints that are specific to a domain. An example in our domain questionnaire would be requiring that the maximum number of questions per questionnaire be 40 questions irrespective of how they are grouped into blocks. These

²For notational simplicity, we equate the scopes with the rule instances.

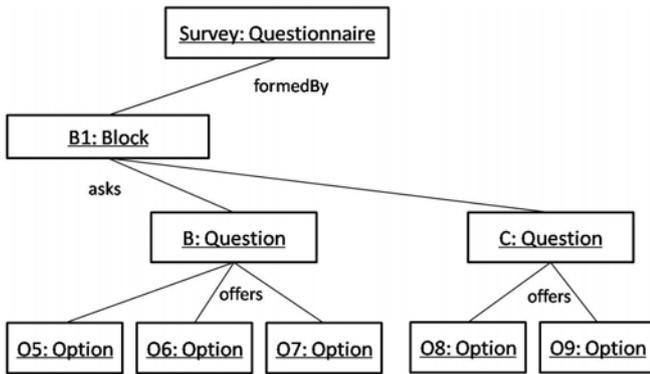


Fig. 6. Feature G, instance of Questionnaire metamodel

domain-specific conformance rules are treated identically to those generated from well-formedness rules.

In summary, the conformance of a model to a metamodel can be checked using incremental consistency checking where the constraints rules are: *i*) rules generated from the well-formedness rules of the meta-metamodel that apply in a metamodel, *ii*) domain-specific constraints defined for the metamodel.

B. Conformance during Model Composition

Let us describe now how model conformance is checked during model composition. Consider the feature G in Figure 6. This feature also has a block B1 with Question B with three new Option answers, and another Question C with its two Option answers. It is clear from our description above that this feature conforms to its metamodel as all the conformance rules instances are valid. Regardless of the technology used [25], model composition can be seen as applying a successive set of changes to an existing model. In our example, the composition of this second feature G to feature F in Figure 4 adds three new options to question B, and a new question C with its options.

The first step prior to start composition is cloning copies of the features involved and their scopes. On these copies will composition and conformance checking be performed. This step is necessary because features can be used to compose different products. In FOMDE, features are composed hierarchically starting from the root element. Elements that have the same name and type at the same hierarchical level are composed together, elements that do not have a corresponding matching element are copied along hierarchically. In our example, elements Survey, B1 and B from feature G have a matching element in feature F thus they will compose hierarchically. The remaining elements in feature G are copied along at their corresponding level.

As composition proceeds, a rule instance is re-evaluated if a change in its scope elements is detected. Additionally, if model elements are deleted or new ones are added, new rule instances can be removed or created. The result of composing our two features is depicted in Figure 7 with their corresponding consistency rule scopes. For visual simplicity,

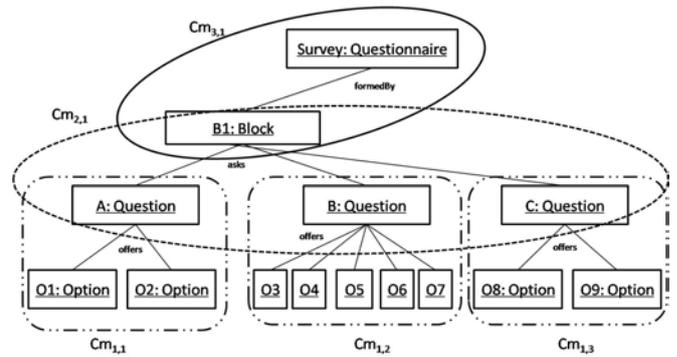


Fig. 7. Composed feature with scopes

the types Option of the objects of question B are omitted in the figure.

In our example, the addition of options O5, O6, and O7, causes a re-evaluation of rule instance $cm_{1,2}$. Recall that rule cm_1 checks the aggregation between Option and Question, such that each question has from two to four possible answers. Therefore, this instance re-evaluation detects a violation of this rule because Question B now has five available options. It is important to notice that this violation is signaled as soon as option O7 is added. This immediate notification allows the developer to take any corrective actions deemed necessary. A possibility is backtracking composition to trace the source of the non-conformance.

Continuing with the composition, the addition of the new Question C creates a new instance $cm_{1,3}$ whose evaluation meets the conformance rule. Because there was a change in the scope of $cm_{2,1}$ resulting from the addition of a new question, this instance is also re-evaluated. Recall that rule cm_2 checks aggregations between Block and Question such that a block has at least one question. Thus the re-evaluation of $cm_{2,1}$ does not detect any inconsistencies. In conclusion, the only non-conformance to the metamodel of the composed features is because five options are available for Question B.

Summarizing, conformance during model composition follows the same process described in the previous section. The insight here is considering the composition of a feature with another as applying a set of finer-grain model changes to another model.

C. Conformance during Metamodel Composition

The same principles of incremental consistency checking are applicable for composing metamodels. To illustrate that, first consider Figure 8 that shows a metamodel feature that has an aggregation for Block to itself, and a navigable association from Block to a new class Scale. We will compose this metamodel with Base metamodel in Figure 3.

Let us explain how incremental consistency works when metamodels are composed. The first consideration to keep in mind is that a metamodel is in itself an instance of a meta-metamodel. Thus a metamodel can be viewed as a set of instances of meta-meta-classes. For example, using Ecore

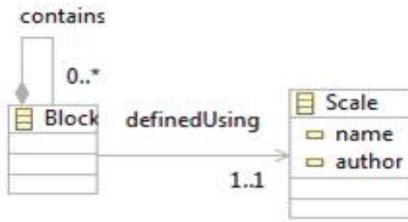


Fig. 8. Scale metamodel and feature

[23], Figure 9 shows a simplified view of the questionnaire metamodel in Figure 3. A package (meta-metaclass EPackage) aggregates zero or more classes (meta-metaclass EClass) in an aggregation called eClassifiers. In turn, each EClass instance aggregates its attributes (meta-metaclass EAttribute) and its references to other classes (meta-metaclass EReference). Note that these references are the aggregations between the metaclasses in Figure 3. For example, the EReference formedBy in EClass Questionnaire corresponds to the formedBy aggregation between metaclasses Questionnaire and Block³.

Using this perspective of considering metaclasses as instances (model elements) of the meta-metamodel, we can apply exactly the same process we followed for checking conformance with model composition. First, clone copies of the metamodels and their scopes are made to apply composition and conformance checking on them.

Two conformance rules that check aggregation, but now at the metamodel level, can be generated. We use suffix mm to denote these rules as they are now defined in terms of metamodel elements as follows:

- cmm_1 = Aggregation rule with A is EPackage and B is EClass
- cmm_2 = Aggregation rule with A is EClass and B is EStructuralFeature

³For simplicity the endType of the EReference is not depicted. For instance in the case of formedBy this type is metaclass Block.

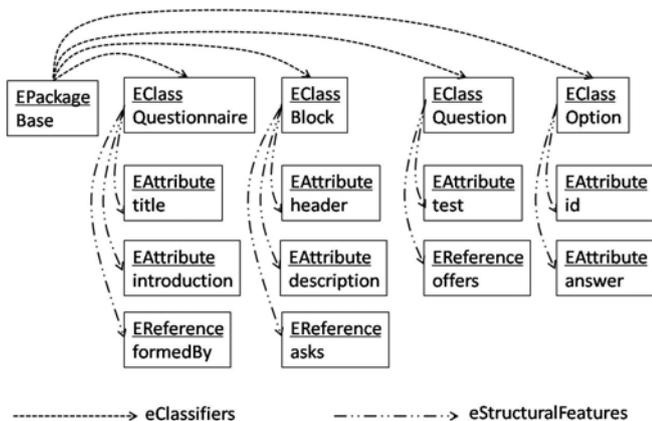


Fig. 9. Simplified metamodel view in terms of metaclasses

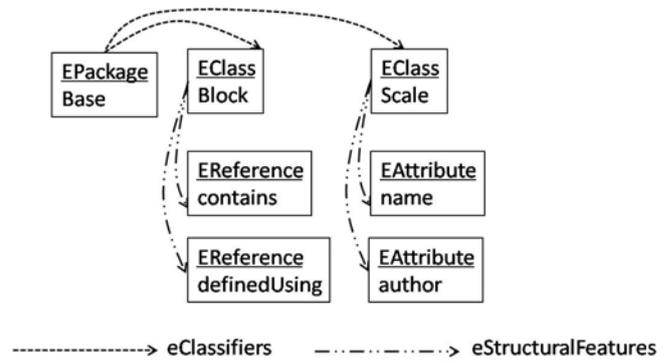


Fig. 10. Simplified metamodel view of Scale

In Ecore, EStructuralFeature is an interface implemented by both EClass and EReference. Furthermore, we can now identify the following instances of these rules and their corresponding scopes in Questionnaire metamodel:

- $cmm_{1,1} = \{Base, Questionnaire, Block, Question, Option\}$
- $cmm_{2,1} = \{Questionnaire, title, introduction, formedBy\}$
- $cmm_{2,2} = \{Block, header, description, asks\}$
- $cmm_{2,3} = \{Question, test, offers\}$
- $cmm_{2,4} = \{Option, id, answer\}$

Using this same perspective, the metaclasses view of Scale metamodel is depicted in Figure 10. Metamodel composition, performed along the lines illustrated in previous section, modifies the scopes of the rule instances $cmm_{1,1}$ and $cmm_{2,2}$ (changes are underlined) and creates a new rule instance $cmm_{2,5}$ as follows:

- $cmm_{1,1} = \{Base, Questionnaire, Block, Question, Option, Scale\}$
- $cmm_{2,2} = \{Block, header, description, asks, contains, definedUsing\}$
- $cmm_{2,5} = \{Scale, name, author\}$

Consequently instances $cmm_{1,1}$ and $cmm_{2,2}$ need to be re-evaluated, and instance $cmm_{2,5}$ evaluated for a first time. In this case these instances do not cause any conformance violations as they meet the constraint given that a package can have zero or more classes, and a class can have zero or more attributes and references. In other words, the composed metamodel conforms to the meta-metamodel.

Despite of not causing any inconsistency, the changes in the metamodel can still trigger the generation of new rule instances as new metaclasses can be added. In this example, the addition of EReference contains causes the generation of a new instance cm_4 of the aggregation rule with A is Block and B is Block. This generation in turn triggers a search for instances of cm_4 at the model level. In our model composition examples we have no such case, so the checking process stops there.

In summary, checking conformance of metamodel composition follows the same process as the case of model composition

but with the additional step that changes at the metamodel level can trigger the generation of new rule instances or the re-evaluation of existing instances at the model level.

D. Potential benefits

Based in our experience, early conformance checking of features by means of incremental consistency checking can offer three major advantages when compared to that batch checking:

- Consistency of modeling artifacts throughout the entire development process, including their correctness and well-formedness [26].
- Earlier identification of inconsistencies.
- Traceability of the origin of the inconsistency.

IV. RELATED WORK

There is extensive research on models, model composition and SPL. In this section we shortly present those pieces of research that most closely relate to our work.

Safe composition is the guarantee that programs composed according to the product line constraints are type safe [27], i.e. they do not have undefined elements to structural elements such as classes, methods, and fields. Contrary to this paper that focuses on checking conformance of a given product, safe composition focuses on validating properties for all members of a product line. Our recent work has shown how to use UML consistency rules as the constraints to validate safe composition in UML-based SPL [28].

There are several approaches and technologies to perform model composition [25]. Only a few are specific to SPL. Trujillo et. al motivate the need of realizing variability not only at model level but also at metamodels and model transformations [18]. FeatureHouse uses model superimposition to compose basic UML models at the XMI level [29], and MATA uses graph transformations as composition mechanisms for UML models [30]. However, in these two approaches it is unclear how (if at all) conformance checking is performed.

Another approach implements feature composition uses Maude, a high-performance logical framework [31]. In this work, feature composition is expressed in terms of rewrite rules. Conformance checking becomes reduction according to the rewrite rules, which in our context effectively are our conformance rules. In other words, if a composed model reduces to a canonical form (one that cannot be further reduced), the model conforms, otherwise an error can be detected. We have not investigated how this approach could be tailored to represent arbitrary domain-specific constraints. This issue is part of our future work.

V. OPEN QUESTIONS

In this section we sketch some of the open questions we identified in our work, as such, they are venues for our future work.

Living with inconsistencies. In this paper we assumed that the composition of models and metamodels should at any-time conform respectively to their metamodels and metametamodels. However, there may be intermediate stages during

composition at which this assumption may not hold, but still yield a conforming result at the end. For example, if the composition paradigm used were non-monotonic (permitting to remove model elements) and our features F and G were composed with a third feature that removed one of the options of `Question B`, the result would be a conforming model in despite of the partial composition of F and G being non conformant. This type of inconsistency is *tolerable* as composition may potentially be able to "fix" it. There may be also cases where an inconsistency cannot be remedied. A case from our `Questionnaire` metamodel would be a feature that contains two `Option` objects without any associations. This type of inconsistency is *intolerable* because for this feature to be composable the `Question`, the `Block`, and the `Questionnaire` the options belong to must be also defined. Thus, living with inconsistencies [32], [33] (tolerating some of them) also plays an important role in our work. Characterizing, identifying and managing both types of inconsistencies may have an impact on how we define and implement our conformance rules.

Impact of a change. We expect changes not to be isolated. Constraint instances may have complex relationships amongst them in such a way that a single change may trigger a cascade of inconsistencies for which subsequent fixes may be required. Efficiently determining the impact of a change and computing an order in which to fix the triggered inconsistencies may be a crucial point for our approach to adequately scale.

Consistency at other development stages. Conceivably, there are other scenarios where changes can also occur and thus conformance checking may become necessary. An example is when features, either models or metamodels, themselves change as consequence of changes in the requirements. These changes may themselves trigger conformance checks of the modified models and metamodels. Another possible scenario is when concrete products evolve and such changes must be propagated back into the SPL architecture and its features. In summary, we plan to study all other possible scenarios where conformance checking may be needed and evaluate the applicability of incremental consistency to them.

Consistencies between feature artifacts. Our paper focused on checking conformance within an artifact type, namely models or metamodels. However, it is common that a feature involves more than one artifact type, such as code, models, XML files, script files, etc. Thus it is important to keep consistency amongst the elements of a feature. Our recent work has started to address this issue with UML artifacts [28].

Evolution direction. In our work, when changes occur at the metamodel level, other changes can be triggered down at the model level. However, it is conceivable that changes may flow in the opposite direction. This means that a change committed at the model level imposes changes at the metamodel level which in turn may trigger other changes at other instances of the metamodel.

Safe composition. The work presented in this paper focuses on checking the conformance of one concrete product. The goal of safe composition is the verification that certain

constraints are met in all the possible configurations (allowable combinations of features) of a product line. However, because SAT solvers are used for this validation, there may be scalability issues as the size of feature models, the types of constraints, and number artifact types increase. Knowing those potential limitations could help provide guidance on how to extend safe composition to address the above mentioned open questions.

Consistency between variability space and solution space. This goes a step beyond safe composition by not only detecting violation of the variability at the implementation level but also in attempting to keep consistent variability defined in a feature model with its realization across multiple artifacts. We believe that the intensive ongoing research in formal analysis of feature models can provide a foundation to address this question [34]–[36].

VI. CONCLUSIONS AND FUTURE WORK

In this paper we drew a connection between FOMDE and incremental consistency checking. A crucial need in FOMDE is checking conformance in different scenarios: model to metamodel, during model composition, and during metamodel composition. We showed that the underlying principles of incremental consistency checking are applicable for checking conformance in these three scenarios. The key is using as consistency rules the conformance rules that are generated from meta-metamodel well-formedness specifications or constraints that are domain-specific.

As a first step, we plan to develop a metamodel-independent framework to facilitate the specification of constraint rules and their subsequent generation. We will use this framework to evaluate our approach in industry-motivated cases studies and address the identified open questions.

ACKNOWLEDGMENT

We thank Laura Vozmediano for her help with the questionnaires domain. This research is partially sponsored by the Austrian FWF under agreement P21321-N15. This work is co-supported by the Spanish Ministry of Science and Innovation, under contracts TIN2008-06507-C02-01 and TIN2008-06507-C02-02.

REFERENCES

- [1] J. Bézivin, "On the unification power of models," *Software and System Modeling*, vol. 4, no. 2, pp. 171–188, 2005.
- [2] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [3] K. Pohl, G. Bockle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
- [4] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
- [5] D. S. Batory, R. E. Lopez-Herrejon, and J.-P. Martin, "Generating product-lines of product-families," in *ASE*. IEEE Computer Society, 2002, pp. 81–92.
- [6] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE TSE*, vol. 30, no. 6, 2004.
- [7] P. Zave, "Faq sheet on feature interaction," <http://www.research.att.com/pamela/faq.html>.
- [8] S. Trujillo, D. Batory, and O. Diaz, "Feature Oriented Model Driven Development: A Case Study for Portlets," in *ICSE*, 2007.
- [9] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A framework for integrating multiple perspectives in system development," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 1, pp. 31–57, 1992.
- [10] A. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency handling in multiperspective specifications," *IEEE Trans. Software Eng.*, vol. 20, no. 8, pp. 569–578, 1994.
- [11] B. Nuseibeh, J. Kramer, and A. Finkelstein, "A framework for expressing the relationships between multiple views in requirements specification," *IEEE Trans. Software Eng.*, vol. 20, no. 10, pp. 760–773, 1994.
- [12] "Unified Modeling Language (UML)," 2008, <http://www.uml.org>.
- [13] D. S. Batory, M. Azanza, and J. Saraiva, "The objects and arrows of computational design," in *MoDELS*, ser. Lecture Notes in Computer Science, K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, Eds., vol. 5301. Springer, 2008, pp. 1–20.
- [14] G. Freeman, D. S. Batory, and R. G. Lavender, "Lifting transformational models of product lines: A case study," in *ICMT*, ser. Lecture Notes in Computer Science, A. Vallecillo, J. Gray, and A. Pierantonio, Eds., vol. 5063. Springer, 2008, pp. 16–30.
- [15] E. Uzuncaova, D. Garcia, S. Khurshid, and D. S. Batory, "A specification-based approach to testing software product lines," in *ESEC/SIGSOFT FSE*, I. Crnkovic and A. Bertolino, Eds. ACM, 2007, pp. 525–528.
- [16] F. I. Anfurrutia, O. Diaz, and S. Trujillo, "On the Refinement of XML," in *International Conference on Web Engineering ICWE*, 2007.
- [17] J. White, J. H. Hill, J. Gray, S. Tambe, A. S. Gokhale, and D. C. Schmidt, "Improving domain-specific language reuse with software product line techniques," *IEEE Software*, vol. 26, no. 4, pp. 47–53, 2009.
- [18] S. Trujillo, A. Zubizarreta, X. Mendiola, and J. de Sosa, "Feature-oriented refinement of models, metamodels and model transformations," in *FOSD*, ser. ACM International Conference Proceeding Series, S. Apel, W. R. Cook, K. Czarnecki, C. Kästner, N. Loughran, and O. Nierstrasz, Eds. ACM, 2009, pp. 87–94.
- [19] F. Lucas, F. Molina, and A. Toval, "A systematic review of UML model consistency management," in *To appear Information and Software Technology*, 2009.
- [20] M. Usman, A. Nadeem, T.-H. Kim, and E.-S. Cho, "A survey of consistency checking techniques for uml models," in *Advanced Software Engineering and Its Applications, 2008. ASEA 2008*, 2008, pp. 57–62. [Online]. Available: <http://dx.doi.org/10.1109/ASEA.2008.40>
- [21] A. Egyed, "Instant consistency checking for the uml," in *ICSE*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 2006, pp. 381–390.
- [22] —, "Fixing inconsistencies in uml design models," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 292–301.
- [23] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed. Addison-Wesley Professional, 2008.
- [24] OMG, "Meta Object Facility (MOF)," 2010, <http://www.omg.org/mof>.
- [25] C. Jeanneret, "An analysis of model composition approaches," Master's thesis, Ecole Polytechnique Federal de Lausanne, 2008.
- [26] F. Heidenreich, "Towards systematic ensuring well-formedness of software product lines," in *FOSD*, ser. ACM International Conference Proceeding Series, S. Apel, W. R. Cook, K. Czarnecki, C. Kästner, N. Loughran, and O. Nierstrasz, Eds. ACM, 2009, pp. 69–74.
- [27] S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook, "Safe composition of product lines," in *GPCE*, C. Consel and J. L. Lawall, Eds. ACM, 2007, pp. 95–104.
- [28] R. E. Lopez-Herrejon and A. Egyed, "Detecting inconsistencies in multi-view models with variability," submitted for publication.
- [29] S. Apel, C. Kästner, and C. Lengauer, "Featurehouse: Language-independent, automated software composition," in *ICSE*. IEEE, 2009, pp. 221–231.
- [30] P. Jayaraman, J. Whittle, A. Elkhodary, and H. Gomaa, "Model Composition and Feature Interaction Detection in Product Lines using Critical Pair Analysis," in *MoDELS*, 2007.
- [31] R. E. Lopez-Herrejon and J. E. Rivera, "Realizing feature oriented software development with equational logic: An exploratory study," in *JISBD*, A. Vallecillo and G. Sagardui, Eds., 2009, pp. 269–274.
- [32] R. Balzer, "Tolerating inconsistency," in *ICSE*, 1991, pp. 158–165.
- [33] S. Fickas, M. Feather, and J. Kramer, "Living with inconsistency. icse workshop, boston, usa," 1997.

- [34] D. Benavides, A. R. Cortés, D. S. Batory, and P. Heymans, “First international workshop on analysis of software product lines (aspl’08),” in *SPLC*. IEEE Computer Society, 2008, p. 385.
- [35] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. R. Cortés, “Automated diagnosis of product-line configuration errors in feature models,” in *SPLC*. IEEE Computer Society, 2008, pp. 225–234.
- [36] D. Batory, “Feature Models, Grammars, and Propositional Formulas,” in *Proceedings of the International Software Product Line Conference (SPLC)*, 2005, pp. 7–20.
- [37] *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*. IEEE Computer Society, 2008.
- [38] S. Apel, W. R. Cook, K. Czarnecki, C. Kästner, N. Loughran, and O. Nierstrasz, Eds., *Proceedings of the First International Workshop on Feature-Oriented Software Development, FOSD 2009, Denver, Colorado, USA, October 6, 2009*, ser. ACM International Conference Proceeding Series. ACM, 2009.

The CVM Framework — A Prototype Tool for Compositional Variability Management

Andreas Abele (*Continental AG, DE*), Rolf Johansson (*Mentor Graphics, HU*), Henrik Lönn (*Volvo Technology, SE*),
Yiannis Papadopoulos (*University of Hull, UK*), Mark-Oliver Reiser (*Technische Universität Berlin, DE*),
David Servat (*Commissariat à l’Energie Atomique, FR*), Martin Törngren (*Kungliga Tekniska Högskolan, SE*) and
Matthias Weber (*Technische Universität Berlin, DE*)

Abstract—This article announces the first public release of an experimental research tool for variability management, called “CVM framework” and provides an overview of the tool’s capabilities and architecture.

I. INTRODUCTION

Over the past few years, an experimental variability management tool was developed at Technische Universität Berlin, called “CVM framework”. It was mainly intended for the evaluation of several research approaches developed in a number of industrial cooperations, esp. with Daimler AG and Carmeq GmbH / Volkswagen AG and the European research project ATESS2 (the name “CVM” originated from one of these approaches called “compositional variability management” [12]).

This paper announces the first public release of the tool [4]. We provide an overview of the tool’s capabilities (Section II) and architecture (Section III) and briefly report on recent applications (Section IV) before concluding with a discussion of related work and an outlook.

II. KEY CAPABILITIES OF THE FRAMEWORK

The key functionality of the tool can be divided into the four areas of (a) basic feature modeling and configuration, (b) multi-level feature modeling, (c) configuration links, and (d) configuration graphs.

A. Feature Modeling & Feature Configuration

The CVM framework is heavily based on feature modeling [9], [5]. It was an important design goal to not reinvent the wheel by defining a novel feature modeling approach, but instead making the tool as compatible as possible to classic feature modeling techniques from the literature. Its main basis was cardinality-based feature modeling of Czarnecki et al. [6], but with several extensions from other authors. We cannot provide a comprehensive list here, but an excellent overview of feature modeling techniques with a multitude of further references can be found in [14]. The editing support for feature modeling comprises:

- essential feature editing capabilities (creating and deleting features, moving features within the feature tree hierarchy, ...)
- optional, mandatory and cloned features
- feature groups
- feature inheritance (often called “feature references” in the literature)

- feature links (for simple dependencies between a single start and a single end feature, such as “needs” and “excludes”)
- feature constraints (for more complex dependencies beyond the expressiveness of feature links)
- parameterized features (sometimes called “feature attributes” in the literature)
- explorer-style tree views as well as graphical views of the feature models
- editing of configurations of feature models
- a simple type system for checking the validity of values of parameterized features during configuration
- customizable user attributes for project-specific meta-information (which can be attached to most elements)

Figure 1 shows a simple feature model in both the tool’s explorer-style model editor and the graphical feature diagram editor. For feature diagrams, the implementation follows the common model/diagram pattern: for a single feature model, several feature diagrams may be defined that provide different views on the model. This means that some parts of the model, i.e. some features, may show up in a particular diagram while others do not, and if they appear in more than one diagram, all the diagrams show the same model objects.

As mentioned above, a configurator is provided to support configuration of feature models, i.e. to select and deselect their features and provide values for parameterized features if they are selected. Figure 2 shows this for the feature model from Figure 1. Optional features are presented with a check box (e.g. `CruiseControl` or `Radar`). The check box has three states: undecided, included and excluded. For example, feature `CruiseControl` is currently undecided as indicated by the small question mark in the check box. Optional features that are in a feature group of cardinality [1] are presented with radio buttons to indicate that they are mandatory alternative (e.g. `Standard` and `Adaptive` below the cruise control). Cloned features with a maximum cardinality greater than 1 (e.g. `Wiper[0..2]`) are special in that they cannot be selected or deselected. Instead, instances have to be created for them. To achieve this, the user right-clicks on the cloned feature and selects “Create instance ...” from the context menu. In the example, two instances of `Wiper` were created: `frontWiper` and `rearWiper`. The value of a parameterized feature can be set by right-clicking the feature and then selecting

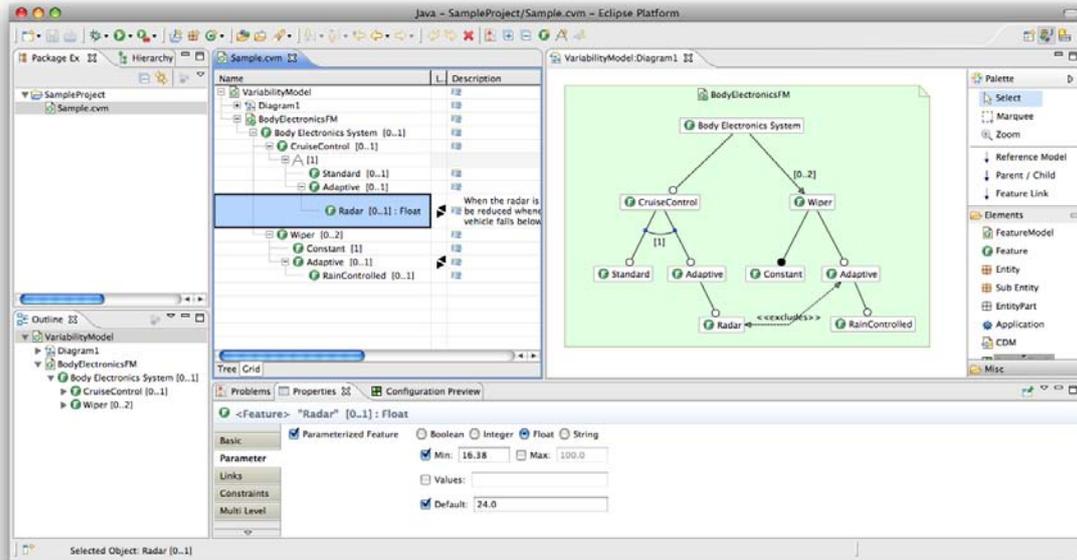


Figure 1. Feature modeling in CVM: the model editor (left) and the diagram editor (right).

“Set Value ...” from the context menu. In the example, Radar was not yet supplied with a value which is indicated by the label “<undefined>”.

B. Multi-Level Feature Modeling

Multi-level feature modeling is an approach to pragmatically manage several related product lines as a global, composite product family without introducing a rigid product line infrastructure on the global level [13]. It is supported by CVM through the following core functionalities:

- defining reference feature models
- propagating features from a referring model to a reference model and vice versa
- finding deviations in a referring model with respect to its reference model and determining the conformance state of a referring model

C. Configuration Links

Configuration links are a concept for defining a relation between two or more feature models with respect to their configuration. In other terms, a configuration link defines how to configure one or more *target* feature models depending on a given configuration of one or more *source* feature models. With this information, it is then possible to automatically derive configurations of the target feature models whenever configurations of the source feature models are provided. This concept can be used to manage variability within a complex product line by (a) defining orthogonal views on feature models and by (b) consistently managing the variability in component hierarchies, called *compositional variability management*, [12], [11].

A configuration link is defined as a set of so-called *configuration decisions*; each such configuration decision represents a single, conditional rule on how to configure

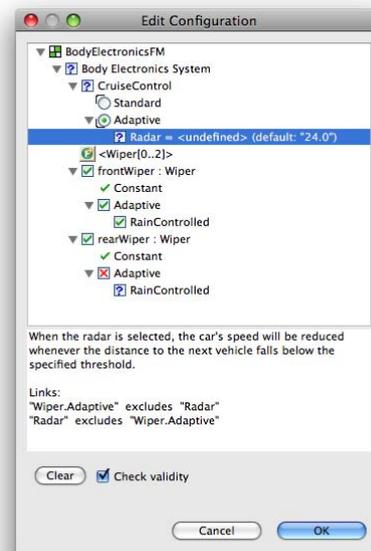


Figure 2. The CVM configurator.

the target feature models, e.g. “if feature USA is selected in the source feature model, then select feature CupHolder in the target feature model”.

Providing prototypical tool support for the evaluation of this approach was the main motivation and driver for implementing the CVM framework tool. Key functionalities for supporting configuration links are:

- essential editing of configuration links and their configuration decisions (creation, deletion, etc.)
- exploring configuration decisions and their impact on the target configuration(s) comfortably, including special markers that highlight the areas where con-

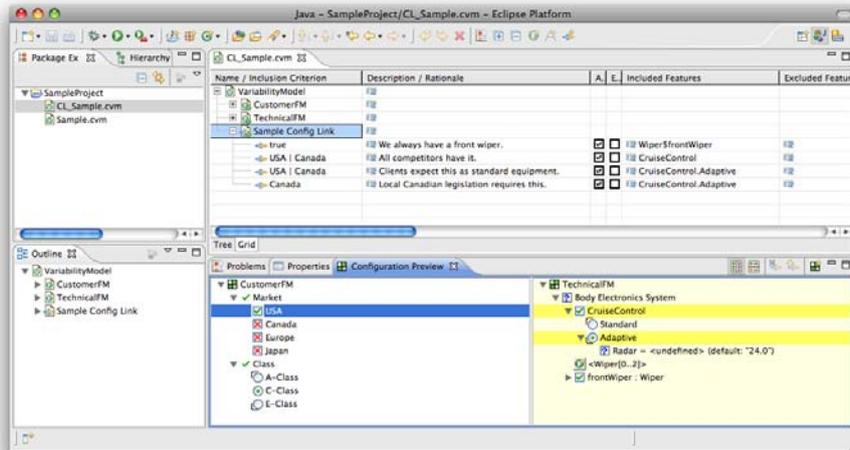


Figure 3. Editing support for configuration decisions with the configuration preview (here shown in “test-drive” mode).

figuration decisions affect the target configurations

- testing the interaction of several selected configuration decisions or a complete configuration link seamlessly while editing is in progress (“test-drive mode”)
- basic supportive analyses to spot contradictions between the configuration decisions within a single configuration link
- automatic configuration derivation, i.e. application of a configuration link on existing configurations of the link’s source feature models, resulting in one or more target configurations

In addition to manually editing the configuration rule captured in a configuration decision, the tool provides means to conveniently edit this information in a special view, called “Configuration Preview”, as presented in Figure 3. The left side of the configuration preview shows configurations of all *source* feature models of the configuration link while the right side shows configurations of the *target* feature models.

This preview has two distinct modes of operation: an *editing mode* that allows to edit configuration decisions and a *test-drive mode* to test the current configuration definition by experimentally configuring the source feature models (left side of the preview) and verifying that the automatically generated configuration of the target feature models (right side of the preview) is correct.

D. Configuration Graphs

Several configuration links can be combined by using the target feature models of a first configuration link as source feature models to a second configuration link, and so on. This way, it is possible to form chains and networks of inter-related feature models, called *configuration graphs* (or more precisely, directed acyclic graphs, in which nodes represent feature models and the edges are realized by configuration links, [11]). A configuration of such a graph, called a *graph configuration*, is a set of ordinary feature

model configurations, one for each feature model / node in the corresponding graph. The tool supports:

- definition and management of configuration graphs of arbitrary complexity
- support for reuse of configuration graphs with a class/instance concept
- editing and automatically deriving graph configurations

It may seem as if configuration graphs are merely a consecutive application of configuration links. However, sophisticated modeling elements and tool support was required to support a feasible management of such configuration graphs and graph configurations, which is the reason for treating these capabilities in a dedicated section, here.

E. Textual Variability Specification

Most variability modeling supported by the framework can be performed on a textual level by way of the built-in *Variability Specification Language (VSL)*. Figure 5 shows the feature model from Figure 1 defined as a textual VSL specification. A single VSL file can contain several feature models, configuration links and configuration graphs. As can be seen, the syntax was inspired by programming languages such as Java, but several optimizations were introduced to better fit the application area of variability specification. For example, all names such as those of features may contain special characters or white-space, which is simply achieved by enclosing them in quotation marks (e.g. "Body Electronics System"). The tree hierarchy in a feature model can be specified using a straightforward notation: each feature may be followed by a comma-separated list of child-features in brackets; each child feature may also have a list of children, and so on. Cardinalities can be specified following the feature name (e.g. Wiper[0..2]); the default cardinality is [0..1], i.e. optional. A cardinality without a name denotes a feature group (e.g. in Figure 5 below feature CruiseControl).

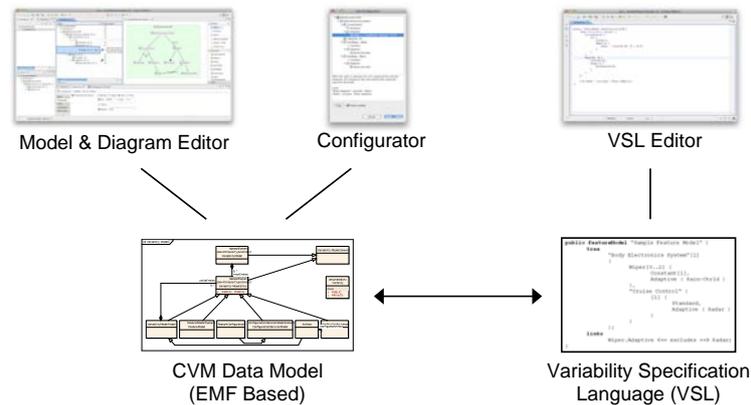


Figure 4. The five main constituents of the CVM framework.

III. CONSTITUENTS AND ARCHITECTURE

The CVM framework is divided into five main constituents, which are illustrated in Figure 4.

Data Model. The data model for variability management is the core of the entire framework. It provides the functionality to programmatically manage variability-related information in memory, for example setting of an object's values, management of bidirectional associations and containment, event notification. In addition, it supports saving and loading the data to/from an XMI file.

Variability Specification Language (VSL). A formal language to textually specify all variability-related information supported by CVM. A parser is provided that transforms VSL specifications to instances of the data model. Also the reverse direction, exporting information from the data model to VSL, is supported.

Model Editor. This is the main editor of CVM and provides a means to interactively navigate and manipulate feature models and other variability-related information.

Diagram Editor. Closely connected to the model editor is the diagram editor for visual editing of feature diagrams and configuration graphs.

Configurator. A configurator is provided to create and edit configurations of feature models. It supports partial configurations with undecided states and management of entire networks of feature model configurations, called graph configurations (cf. Section II-D).

VSL Editor. A text editor with specific support for VSL (e.g. syntax highlighting)

The framework is designed as an Eclipse plug-in. The data model is based on the Eclipse Modeling Framework EMF [3], i.e. an EMOF meta-model of all information entities of CVM was used to generate the Java code of the data model with the EMF code generator. In addition, several manual adaptations and extensions were required. The graphical editing functionality was implemented using the Graphical Editing Framework GEF [7].

IV. APPLICATIONS

The CVM framework has been applied in several smaller industrial experiments, mainly at Daimler AG and Carmeq GmbH / Volkswagen AG. In addition, it has been used

in lectures and various student projects at Technische Universität Berlin. Its main application, however, is in the European FP7 project ATESS2 [2], where it forms an integral part of the project's tool platform. ATESS2 is aimed at defining a comprehensive architectural description language for the automotive domain, called EAST-ADL2, comprising artifacts on several abstraction layers, thus providing a seamless transition from early development phases (features and requirements on vehicle level), via intermediate levels (functional analysis architecture and design architecture) down to implementation level.

The variability modeling approach of ATESS2 is heavily based on feature modeling. One or more feature models are used on vehicle level to define the variability of the complete system, i.e. the vehicle, on a high level of abstraction: in the so-called core technical feature model the system's global variability is defined with a technical perspective; other vehicle-level feature models can be added to realize orthogonal views on this technical variability, such as a marketing-driven packaging of variability for immediate end-customer configuration. Configuration links are used to realize the mapping of these orthogonal views onto the core technical feature model, thus enabling an automatic derivation of a configuration of the core technical feature model from any given configuration of the orthogonal views.

In addition, EAST-ADL2 supports variability within the functional analysis architecture and design architecture. This means the artifacts on these levels can be defined in variable form, mainly by marking individual elements as optional. Configuration graphs, as presented in Section II-D above, can then be used to manage the variability in these artifacts all across the system's component hierarchy. In the end, CVM's graph configuration functionality then provides a means to derive an entire system configuration from given configurations of the vehicle level feature model(s).

EAST-ADL2 and CVM are also being applied in the European project HAVEit [8] to model the architecture of various advanced driver assistance functions, such as an automatic queue assistance (AQuA) and a temporary auto pilot (TAP).

```

1  featureModel BodyElectronicsFM {
2    "Body Electronics System" (
3      CruiseControl (
4        [1] (
5          Standard,
6          Adaptive (
7            Radar : float[16.38..*] = 24.0
8          )
9        )
10     ),
11     Wiper[0..2] (
12       Constant[1], // mandatory
13       Adaptive ( RainControlled )
14     )
15   );
16
17   link Radar < excludes > Wiper.Adaptive;
18 }

```

Figure 5. A sample VSL specification.

V. RELATED WORK — OR: YET ANOTHER FEATURE EDITOR ?

There already exist several tools that provide feature editing functionality, available both as commercial products (e.g. pure::variants [10]) as well as in the form of research prototypes (e.g. CaptainFeature and FMP [1] or XFeature [15]). It therefore only makes sense to come up with a new feature editing tool if the implementation effort is justified by factual necessities.

The primary motivation for implementing the prototypical CVM framework with its own feature editor was the fact, that the feasibility of the underlying concepts, especially of multi-level feature modeling and of configuration links, is closely linked to how they are embedded in the overall feature-editing facility and it was therefore necessary to experiment with different forms of editing functionality. In addition, when building on an existent research demonstrator there would always be the risk that its development is broken off or that the team working on the demonstrator decides to introduce fundamental changes to the tool's architecture or concepts which could lead to irreconcilable conflicts with assumptions made in the own project. Finally, building basic editors for your own newly introduced domain specific methodologies became a lot easier thanks to latest achievements in model driven development, for example the Eclipse Modeling Framework (EMF), the Graphical Editing Framework (GEF), and associated projects.

VI. DISCUSSION AND OUTLOOK

Though not intended for application in mission-critical projects, the CVM framework as presented in this article may be of interest for use in teaching or other experimental applications; also the diagramming functionalities might prove useful when drawing models for presentations or publications. The tool is available for download from [4]. In the future, we will continue to use the tool in our own projects and will extend and adapt it accordingly.

Acknowledgments. The research leading to the CVM tool as presented in this article was partly supported by a research scholarship of Daimler AG and received funding from the European Community's 7th Framework Programme under grant agreement no. 224442.

REFERENCES

- [1] Michal Antkiewicz and Krzysztof Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange (ETX 2004)*, pages 67–72. ACM Press, 2004.
- [2] ATESSST Project Web-Site, 2008. www.atesst.org.
- [3] Frank Budinsky, David Steinberg, Ed Merks, and Raymond Ellersick. *Eclipse Modeling Framework*. Addison Wesley, 2003.
- [4] CVM-Framework Project Web-Site, 2009. www.cvm-framework.org.
- [5] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming*. Addison-Wesley, 2000.
- [6] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practices*, 10(1):7–29, 2005.
- [7] Eclipse Foundation Web-Site, 2008. www.eclipse.org.
- [8] HAVEit Project Web-Site, 2009. www.haveit-eu.org.
- [9] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) – feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute (SEI), Carnegie Mellon University, 1990.
- [10] Pure-Systems GmbH Web-Site, 2008. www.pure-systems.com.
- [11] Mark-Oliver Reiser. Core concepts of the Compositional Variability Management framework (CVM). Technical Report, no. 2009-16, Technische Universität Berlin, 2009.
- [12] Mark-Oliver Reiser, Ramin Tavakoli, and Matthias Weber. Compositional variability. In *Proceedings of the 42nd Hawaii International Conference on System Sciences (HICSS-42)*. IEEE Computer Society Press, 2009.
- [13] Mark-Oliver Reiser and Matthias Weber. Multi-level feature trees – a pragmatic approach to managing highly complex product families. *Requirements Engineering*, 12(2):57–75, Apr 2007.
- [14] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.
- [15] XFeature Feature Editor Web-Site, 2008. www.pnp-software.com/XFeature/home.html.

Conflict Resolution Strategies during Product Configuration

Alexander Nöhrer

Johannes Kepler University
Institute for Systems Engineering
and Automation (SEA)
Linz, Austria
alexander.noehrer@jku.at

Alexander Egyed

Johannes Kepler University
Institute for Systems Engineering
and Automation (SEA)
Linz, Austria
alexander.egyed@jku.at

Abstract—During product configuration, users are prone to make errors because of complexity and lack of system knowledge. Such errors cause conflicts (i.e., incompatible choices selected) and current state-of-the-art configurators require users to undo earlier decisions made or restart the decision process altogether. This paper discusses these and other conflict resolution strategies; even ones that allow users to introduce conflicts and solve them at a later time of their choosing. This is analogous to the notion of living with inconsistencies which is not only tolerated but deemed necessary in modeling. We will discuss that allowing conflicts to exist during the configuration process (living with conflicts) is likewise beneficial during the configuration process because it is easier to resolve conflicts at a later time when the user's intention is better understood (i.e., more input was provided). However, the dilemma with living with conflicts is that traditional reasoning mechanisms become inadequate. For example, it is common during configuration to eliminate choices of future decisions (unanswered questions) based on decisions that have already been provided and we will discuss how to continue doing so in the presence of conflicts. Furthermore, we will discuss that understanding the order (history) of decisions made is beneficial for better resolving conflicts later.

Keywords—Product Line Engineering; Formal reasoning; User Guidance

I. INTRODUCTION

User guidance during product configuration is perceived to be a straightforward activity where a user answers a set of pre-defined questions, usually by selecting among their choices. For example, an online laptop configurator is such a system. It typically has predefined questions for building a laptop, each question with a predefined set of choices (e.g. RAM 4GB or 8GB).

Without detailed expert knowledge, users are confronted with the exponentially complex task of navigating among interdependent choices and their implications without explanations. It is state-of-the-art to support users by asking questions in a predefined order and presenting only those choices of the remaining questions that are still available [1]. For example, after selecting a 32-bit operating system for the laptop, the 8GB RAM choice becomes unavailable. Initially all choices are available, these are then incrementally reduced as the user answers questions (decides on a choice).

Current approaches to product configuration are able to restrict user choices based on decisions already provided, as for instance covered in [2-4] and other commercial and research prototype configurators. However, if the configurator does not support conflicts and a desired choice is already eliminated then the user has only one option: undo previous decisions and trying alternative combinations that might be acceptable *without knowing whether the alternatives will lead to such dead ends again!* For example, once the user encounters that the 8GB RAM choice is not available, then this requires undoing the operating system choice. However, without domain knowledge this would be hard to guess. Moreover, the user might be uncertain as to whether changing the laptop type would resolve the problem also. This leads to exponentially increasing combinations on how to resolve such conflicts. Without good tool support, users will find it very hard indeed to navigate this jungle of questions and choices – particularly, if the user is not an expert user which is the case in most situations.

To help the user in this complex conflict resolution task, for example the works by [5] could help identify those decisions that are in conflict with a desired choice. Only these decisions must be revisited (i.e., undone) which is more efficient than revisiting all decisions. We can think of such an approach as a selective undoing of conflicting decisions to be used at the time where a desired choice is not available. While such an approach reduces the complexity of the problem, it does not avoid the fundamental problem: the user still needs to navigate alternative choices of those questions in the hopes of identifying those that do not eliminate the desired choice. Moreover, in the case where multiple, alternative options exist on how to resolve the conflict; the user has to make a suboptimal decision of which answer to undo. A decision is suboptimal because the undoing would not consider future user decisions for a more optimal reasoning.

To avoid making suboptimal decisions, the alternative is to resolve conflicts at the end – after all questions have been answered. For instance the approach by [6] advocates such an alternative by helping identify conflicting decisions at any point in time, to find the one valid (partial) configuration that closest matches the desired decisions (valid configuration with minimal deviation from the user selected configuration). Such an approach allows users to select conflicting choices. However, the disadvantage of resolving conflicts at the end

is that users then lose the ability to have choices reduced automatically and incrementally as answers are provided – an important feature discussed earlier. This downside exists because existing reasoning engines (e.g., Theorem proving to check for satisfiability also known as SAT solvers [7]) do not readily function in the presence of contradicting information. The user is on his/her own which may be ok for the choices the user wanted to have despite the conflicts, but unnecessary for the other choices where multiple choices would have been acceptable. Moreover, reasoning as it is currently done, ignores the order in which questions are answered (and conflicts are encountered) which we will see later weakens the kinds of analyses we can do.

The ideal solution would be one that allows users to select conflicting choices, however, still supports incremental reasoning, such as the elimination of choices. This requires reasoning in the presence of conflicts. This is not unlike software modeling where reasoning in the presence of inconsistencies is not only tolerated [8] but even advocated as a normal way of life [9]. This paper discusses our approach of living with conflicts during product configuration. For completeness, we describe all possible conflict resolution strategies, from simple ones where living with conflicts is not necessary up to complex ones, which require living with conflicts. *All of them are useful and suitable in different situations of the configuration process. All of them have a right to exist.*

The main contribution of this paper is the ability to reduce the remaining choices of questions despite the presence of conflicts because this aspect is new and novel (i.e., not covered in related work). This is done by conservatively excluding offending decisions from the reasoning core and continuing reasoning with the subset of non-conflicting decisions. Another contribution is the use of the history of decisions made (the order in which questions are answered) as meta information for identifying the offending decisions. In essence, if a user desires a choice and accepts introducing a conflict, then this choice must be more important to the user than some previous decisions. Our approach makes use of this knowledge which is another reason why it is beneficial to still have the ability to reduce choices in the presence of conflicts to deal with multiple conflict situations. The history thus gives insights on what the user's intentions are while he/she is still configuring a product. Depending on these intentions different strategies can be utilized to resolve conflicts.

This paper is structured as follows: In Section II we describe the scenario and problem we address. This is followed by the vision of how we want to tackle the problem in Section III. In Section IV we discuss the state-of-the-art and related work. In Section V we describe in detail how our vision can be realized. The bigger picture is discussed in Section VI. Finally we draw a conclusion and give an outlook to future work in Section VII.

II. SCENARIO AND PROBLEM

During product configuration the preferred working mode is to answer questions by sequentially iterating over features until decisions on all variation points are made.

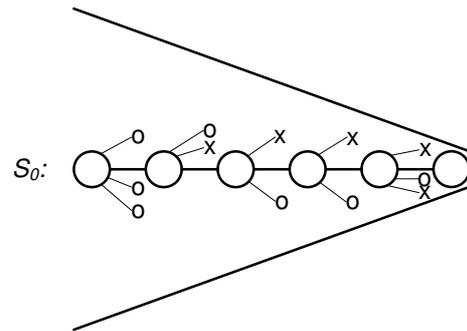


Figure 1. Normal working mode.

Since there are often dependencies among these questions (variation point, or feature), answering a question may affect other questions: it may enable them, it may reduce some of their choices, and it may even answer or eliminate them.

As a real example for a decision-oriented product-line we investigated the laptop configuration system on the DELL website [10] (during the period of February 9th till February 12th 2009) and reverse engineered its product-line model. For illustration purposes we picked three questions about the *Screen Resolution*, the *Screen Size*, and whether a *Webcam* is integrated into the screen. The choices for each question range from more complex enumerations (e.g. *WXGA 1280x800*, *WUXGA 1920x1200*, *XGA 1024x768*, etc. for the *Screen Resolution*) to a simple *yes* or *no* (for the *Webcam*). In this model, many relations between questions exist; relations can also be described as dependencies and constraints respectively. For example since no laptop with a *12.1" Screen Size* and a *WUXGA Screen Resolution* exists, these two decisions would not be compatible and result in a conflict. This and other relations are included in the model together with the choices.

Current state-of-the-art has ample tool support for eliminating choices that are no longer available after a user decision (an answer to a question). In our example this would mean, that the decision *WUXGA* for *Screen Resolution* would eliminate *12.1"* as a *Screen Size* amongst other effects. As a result of these eliminations, a question may even be answered automatically, if all its choices but one are eliminated. Figure 1 S_0 depicts this desired configuration process. It represents a decision tree (flipped sideways) where the big circles represent the decisions made. The leafs at each level represent the alternative choices that were available, but were not chosen by the user (o) or the alternative choices that were eliminated due to earlier decisions and their effects (x) – and thus were not available to the user at the time the question was being answered. The cone indicates the fact that the more questions the users has answered, the closer the user is to an actual configuration. When the last question is answered then the system is fully configured (and all variability is resolved). The cone thus denotes the number of possible configurations, which gets reduced the more questions are answered. As long as users make no decisions that conflict with earlier decisions, this approach works very well and is also well supported in [2, 11].

During the configuration process, users lacking precise system knowledge may discover at one point that a choice they desire to have is no longer allowed anymore because of earlier decisions – meaning the tool eliminated a desired choice because of dependencies. At this point in the configuration process several aspects come into play:

- Users may want an explanation why the choice is no longer available (and perhaps desire to reconsider earlier decisions made).
- Users may want to continue configuring the product and resolve the problem later.

Note, the DELL example is quite analogous to software engineering product lines which we also studied [12]. This is a small instance of a larger engineering challenge. Until a few months ago, DELL laptops could only be configured through a pre-defined order of questions at a predefined starting point (e.g., what type of laptop). Recently, DELL changed this to allow multiple starting points by means of filtering their products according to specific criteria (e.g., memory, screen size). This filtering seems not to work in every case, and not to be exact. This points to software engineers maintaining this feature independently. Also, DELL follows a rather simplistic and unsatisfactory (but easy to implement) notion of living with conflicts. Choices are not eliminated – analogous to the *Continue Manually* strategy discussed later in Section V.A.2a). Clearly, software engineers facing similar problems to the DELL configuration system would benefit from our approach to allow arbitrary starting points and still being able to reason in the presence of conflicts.

III. VISION

Our general vision is to guide and support users but also engineers in situations that cannot be automated. This guidance should be systematic, non intrusive and most importantly allow users the highest degree of freedom, meaning:

- 1) *Users are allowed to make decisions in any order (if so desired).*
- 2) *Users are allowed to resolve conflicts at any time of their choosing.*
- 3) *Users should not be bothered with questions that can be answered through reasoning, meaning the interaction should be reduced to the necessary minimum.*

Of course, the users should continue to benefit from the kinds of automations they expect *despite these freedoms* – for example, still eliminating choices that are no longer available based on previous answers or in reducing the needed user/engineer input to a minimum [12]. Finally, no additional annotations should be required from the user. In other words, the user should not be subjected to providing input that goes beyond what is traditionally done during product configuration.

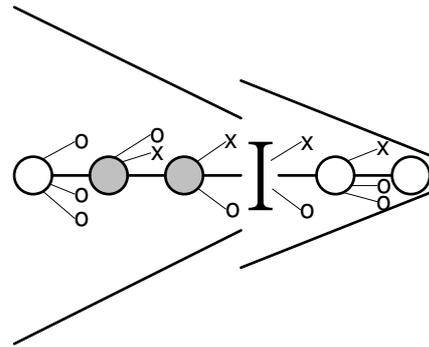


Figure 2. Vision.

To accomplish those goals we are going to tailor a reasoning engine to support reasoning in the presence of conflicts and we are going to use timeline information of the configuration process itself (when happened what). Both concepts have not been used before to resolve configuration errors, to the best of our knowledge. This will enable us to apply different strategies according to different assumptions in combination with the current state in a configuration process – all of which we will describe in detail in the next section.

This work also builds a bridge to the community that works on the problem of living with inconsistencies. Without a similar notion of “living with conflicts”, users would not be allowed to introduce conflicts or would be forced to fix them right away, which are both situations that are not always desirable. The conflict resolution strategies emphasized in this paper are thus strategies where conflicts are tolerated to some degree. This implies that users must be allowed to create conflicting situations.

Our envisioned optimal configuration process is shown in Figure 2, which illustrates the introduction of a conflict into the normal working mode. The **I** denotes the decision that introduced the conflict (inconsistency) and the shaded circles represent the decisions that **I** is in conflict with. Again the cones denote the number of possible choices left to choose from, without violating any constraints (which is analogous to how close the user is to a configuration). As a result of getting closer to a configuration the user’s intentions get more evident and could thus be used as a basis for reasoning. Also if users choose to continue making decisions before resolving the conflicts, the tool should still support them. This support should be realized by eliminating choices of the remaining questions based on the previous answers that are not involved in the conflict. As a consequence, the configuration process is still able to detect new conflicts. Since conflicting answers are no longer used to eliminate the choices of remaining questions, there are likely more choices available. The cone after a conflict is thus bigger again.

In the following, we will discuss resolution strategies for conflicts. As an example for such a conflicting situation we will build on the DELL example discussed earlier - specifically three decisions. The first decision is *12.1"* as a *Screen Size*, the second *XGA 1024x768* as a *Screen Resolution*, and the last *yes* for *Webcam*. The problem with

those three decisions is that DELL does not sell a laptop fulfilling all three decisions. Only laptops with any two of those decisions are sold. The *Vostro 1310* with a *Webcam* and *XGA Screen Resolution*, the *Inspiron Mini 12* with a *Webcam* and *12.1"* as a *Screen Size*, and the *Latitude E4200* with *12.1"* as a *Screen Size* and *XGA Screen Resolution*. So assuming the user has not answered the question about the laptop *Model* yet, the *Latitude E4200* should be excluded because it has no *Webcam*. Furthermore the decision whether it should be the *Vostro 1310* or the *Inspiron Mini 12 Model* (other available models are left out for brevity), would in fact point to the real conflict in the configuration.

IV. STATE OF THE ART

Currently different technologies exist to support the proposed normal working mode; but also to detect, explain and fix conflicts. To enable the normal working mode a few key questions need to be answered:

- 1) *What are the immediate effects of a decision and the elimination of a choice respectively?*
- 2) *What are the ripple effects of a decision and the elimination of a choice respectively?*

Once the system is modeled as a constraint satisfaction problem (CSP), these effects can be calculated with SAT- or CSP-Solvers and used for eliminating conflicting choices [11]. Translating configuration problems/feature models/decision to CSPs is solved and described for example in [13].

As soon as the user leaves the normal working mode and introduces a conflict, other approaches are needed in addition to the checking of satisfiability. With a SAT-Solver conflicts can be detected as a result of the system not being satisfiable anymore, but normally it is not possible to explain where the conflict is coming from or even how many conflicts there are in one configuration. As a consequence different technologies are needed to detect/explain/fix a conflict.

For detecting and explaining conflicts in feature models abductive reasoning can be used as described in [5]. In the UML modeling world Egyed [14] proposed a method for instant checking and as a result detection of inconsistencies (can also be seen as a conflict). Furthermore this approach also implicitly explains why an inconsistency occurred, by pointing to the consistency rules that were violated.

Egyed also proposed methods for fixing inconsistencies in UML models [15, 16]. This technology is able to identify the concrete model elements that are violating a given consistency rules. Applied to our work, model elements are our questions and consistency rules are the relations that trigger conflicts. We believe that this technology can be used to efficiently and correctly identify offending question in case of a conflict. White et al. proposed a method for diagnosing product-line feature models in [6] that in addition proposes minimal solutions to the user. Felfernig et al. also proposed methods for resolving conflicts or as they call it: computing reconfigurations [4]. These technologies are very useful but for our approach we are going to need more than one or several solutions. One single solution or even a few

different solutions almost never actually involve changes in all decisions that are involved in the conflict. Since our vision is to resolve the conflict with new decisions and reason about choices of yet not made decisions, knowing only a few solutions (not all decisions involved in the conflict) is not enough. Including decisions involved in the reasoning about future decisions would bias the results.

In the field of SAT-Solvers minimal solutions can be obtained by searching for a minimal unsatisfiable subset (MUS) of a CNF formula [17]. With the help of such a MUS, decisions that have to be changed to get to a valid configuration can be easily identified. Identifying the maximum number of satisfied constraints, and CNF clauses respectively, in a conflicting configuration is also a possibility. This can be achieved for example with algorithms that solve the Maximum Satisfiability problem (Max-SAT) [18] or solutions to over-constrained Constraint Satisfaction Problems [19]. But again the problem is that not all the decisions involved in the conflict are necessarily identified with these approaches. Nevertheless those technologies are useful for resolving conflicts and are part of the resolving strategies described in the next Section, but not applicable for our envisioned working mode.

As mentioned in the vision in Section III to ensure that users have the highest degree of freedom depends on two things. With regard to the first point that our current work [12] describes how to order questions so that the user input gets minimized without imposing the order onto the user. The order is determined automatically based on effects decisions would have on other questions. This is an incremental process that happens after each decision made by users. In addition to supporting users choosing the next question, engineers also profit from this approach as they do not need to think about an optimal order during modeling. But engineers can influence the outcome of the proposed order through special relations if they wish to do so.

V. CONFLICT RESOLVING STRATEGIES

In this paper, we keep the resolving strategies simple and focused on product configuration, but we believe that the basic strategies discussed here also apply to more general user-guided scenarios. The most important concept that we are using is the history of user decisions, as we mentioned earlier. The pieces of information the sequence reveals are very important to us and need in our opinion to be considered to effectively find solutions on how to fix conflicts. Moreover, automatically made decisions (or eliminated choices) should not be considered as important as user made decisions when taking the sequence and history of decisions into account.

Next we describe the different strategies. First and foremost, we must distinguish two basic cases:

- 1) *No valid configuration exists: this happens when the user configures a product that in this manner does not exist.* Eventually a conflict is found which reveals this problem. This problem can only be fixed by identifying a valid configuration that is "close" to the intent of the

user. The works by White et al. [6] solved this problem with respects to a minimal solution so we do not address it here.

- 2) A valid configuration exists but a conflict was encountered. This is possible if a previous question was answered erroneously or if the configuration process “forced” the user to answer an earlier question without the user understanding the true implications of the choices. As a consequence, a valid solution does exist, albeit some of the questions need to be answered differently.

Note that the two cases are similar in that both find a conflict. The difference is simply in the argument whether an error was made earlier that needs to be fixed (case 2) or whether no error was made and the desired configuration is simply not available (case 1). In case 1, a heuristic needs to be explored to find a “close enough” solution the user might be satisfied with (even if not desired quite as such). In case 2, we have a clear error that must be identified and fixed. No heuristics, no approximations are necessary. We will mostly focus on case 2 in section A below. Case 1 will be briefly discussed in section B.

A. Identifying the Error in a Conflicting Configuration

1) *Fix right away*: At the exact moment the user introduces a conflict into the system by selecting a choice that has been eliminated through some relation; a fixing strategy can be applied to return the configuration to a consistent state immediately. Fixing a conflict right away ensures that the model stays consistent and never contains a conflict (no reasoning in the presence of conflicts is necessary). It is fairly simple to realize and handle with reasoning engines, since the knowledge base stays consistent. Different strategies to fix a conflict right away are illustrated in Figure 3, where the same notation is used as in Figure 2, for sake of brevity the alternatives choices for the decisions and cone are left out. To illustrate the different strategies we again use the example given in Section II. The shaded circles represent the decisions “12.1” as a *Screen Size* and *XGA 1024x768* as a *Screen Resolution*. The **I** represents the decision *Webcam yes*, the other white circles represent other decisions that are not conflicting with each other. Such decisions could be for examples about the CPU, RAM, hard disk, and other laptop components. Next the strategies are described in detail:

a) *Single Undo*: The simplest way to fix a conflict and return to normal working mode is to retract the decision that caused the conflict as illustrated in Figure 3 S_1 . The user is told to try something else instead. Often this is not desired by the user since he/she wants the offending choice. In a more general modeling scenario it could also be the case that a different developer is continuing the work on a model he is not completely familiar with; in such a case *Undo* might not be such a bad idea. In approaches that do not care

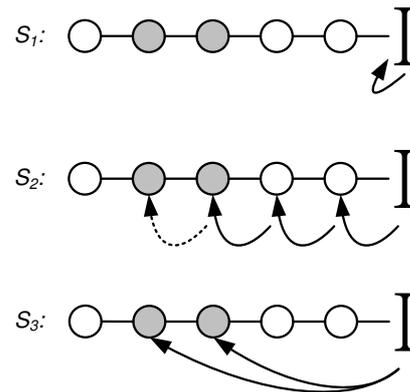


Figure 3. Fix right away strategies.

about the sequence this could also be the solution identified as the minimal solution, since it typically is less effort than changing other conflicting decisions. Applied to our example this would mean retracting the decision *Webcam yes* which certainly would resolve the conflict but may not be desired.

b) *Multiple, Sequential Undo*: Assuming the decision that caused the conflict is important to the user and therefore correct, the problem must be an earlier decision. To find the root of the problem the simplest way is to retract the given decisions until the desired choice for the most recent decision is available. This could also imply retracting decisions that did not contribute to the conflict as illustrated in Figure 3 S_2 (unshaded circles), which is not desirable. In addition to this it could be the case that it is sufficient enough to retract only one of the conflicting decisions. Multiple, sequential undo would retract the most recent one first which could fix the conflict but may not be the desired one. This is also the case in our example, since retracting either the *Screen Size* or the *Screen Resolution* would be sufficient to resolve the conflict, which one gets retracted would depend on the order the decisions were made in.

c) *Selective (Multiple) Undo*: To avoid retracting valid decisions that do not contribute to the conflict, the involved decisions need to be identified. This can be accomplished with abductive reasoning mentioned earlier (Section IV). After the responsible decisions are identified they can be retracted directly as illustrated in Figure 3 S_3 . This approach helps reducing the needed user input compared to the multiple, sequential undo approach (obvious valid decisions do not have to be made more than once). Nevertheless in situations where the desired choice is excluded because of the combination of other decisions, it is not that simple. Retracting one decision or the other could be sufficient, however without further information this cannot be decided automatically. Either all participating decisions or randomly selected among them are retracted, or the user has to be asked which one he/she wants to retract – a question the

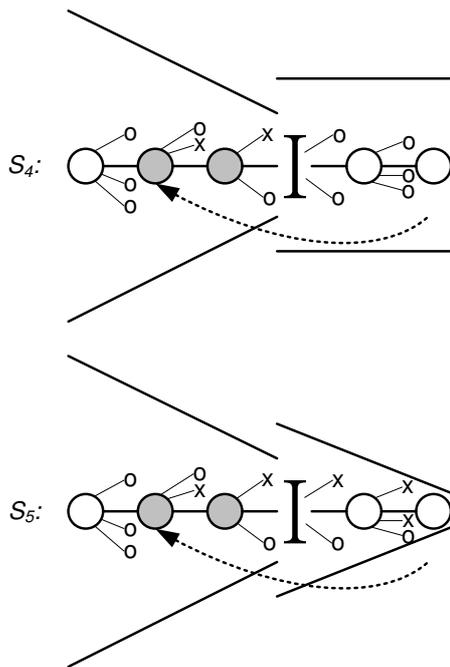


Figure 4. Allow conflicts strategies.

user may not be able to answer correctly! Again this situation can also be found in our example, since retracting either the *Screen Size* or the *Screen Resolution* would be sufficient to resolve the conflict, but this cannot be automatically decided. Selecting one of those decisions randomly or both of them is not a desirable solution. As mentioned above also asking the user about which decision to retract might not be the best thing to do.

The “fix right away” strategies are thus valid but often not desirable. However, in the absence of reasoning in the presence of conflicts, they are the only options available. The following “allow conflicts” strategies present additional options by living with conflicts:

2) *Allow Conflicts*: Instead of fixing a conflict right away, it is more beneficial to let the user answer more/all questions. The more information is collected, the better any reasoning we plan on using works. This additional information may, for example, help in deciding between two alternative options for resolving a conflict. It should be the user’s decision when he/she wants to resolve the conflict. Different strategies to continue the configuration process with a conflict are illustrated in Figure 4, where again the same notation as in Figure 2 is used. These strategies are described in detail here:

a) *Continue Manually*: Since reasoning with conflicts is hard, the simplest way to continue the configuration process is to let the user continue answer questions without such reasoning. However, without reasoning, the user no longer benefits from knowledge of how choices are affected by decisions; i.e., what choices are still valid as illustrated in

Figure 4 S_4 . The negative effect would be that the user is not guided through the remaining questions and as a consequence has to memorize the constraints limiting the product configuration options. This is realistically not possible and unless the user is an expert user, this resolution strategy leads to follow-on conflicts where the user unintentionally makes additional errors. For any reasonably complex system, asking the user to configure a system without automated guidance is a recipe for failure.

b) *Continue with Trust*: Continue with trust means, that assumptions are made on how much certain answers provided by the user can be trusted. For example, the decision that introduced the conflict is a decision that could be trusted to be important to the user – and perhaps even to be final. After all, if a choice is no longer available and the user insists on selecting that choice then the user states that this choice is a “must have”. Obviously, all decisions made earlier that are participating in the conflict could thus be considered less trustworthy. Based on this implicit trust (implied through the order in which questions were answered), the remaining decisions could still be reasoned about – at the very least to conservatively reason about which remaining choices to exclude. With this approach the user is still guided through the remaining questions and informed about decisions that would cause new conflicts as illustrated in Figure 4 S_5 .

The last strategy S_5 *Continue with Trust* thus is the strategy that fulfills all our requirements presented in our vision in Section III. Applied to our example this would mean: Assuming the first decision was about the *RAM*, the second *12.1"* as a *Screen Size*, the third *XGA 1024x768* as a *Screen Resolution*, and the fourth introducing the conflict *yes* for *Webcam*. The next question could be about the laptop *Model*. Since reasoning occurs based on implicit trust the *Latitude E4200* would be eliminated since it has no *Webcam*. In addition the remaining choices *Vostro 1310* or the *Inspiron Mini 12* would help to locate the error, since the *Vostro 1310* is in conflict with *12.1"* as a *Screen Size* too and the *Inspiron Mini 12* is in conflict with the *XGA Screen Resolution*. As required by our vision the requirement that the user can continue making decisions without resolving the conflict is fulfilled. He/she was even supported in doing so through the elimination of choices that would introduce new conflicts and finally the follow up decisions helped in resolving the conflict by pointing to the error.

In addition to the implicit trust described in the *Continue with Trust* section, trust could also be based on user queries: As mentioned in the *Undo* section, in more general scenarios different users can be involved in making decisions. In such cases it could be interesting to ask the user different questions to get a better feeling of what decisions to trust. These questions could range from high-level questions like: *How familiar are you with the given model on a scale from 1 to 5?* to low-level questions like: *Select the decisions that are important to you* (and thus can be trusted) from the list of conflicting decisions. This idea has yet to be elaborated and

tested to work out the details. But we think that it could provide useful information and help to assist users even better. In any case the decisions that can be trusted to be valid are used for a conservative reasoning process just like in the described in the strategy.

B. Identifying a Suitable Alternative

In case the assumption that a solution exist is wrong, meaning that the conflict cannot be resolved with the user satisfied, different strategies have to be exercised to guide the user to an acceptable solution. The details of these strategies have yet to be worked out as well, but the main ideas to get to nearest solution are:

Users can weight their decisions according to the importance to them. To avoid additional user input another possibility would be to somehow automate the weighting according to the decision history. With the help of such weights an optimal compromise could then be found via algorithms like the ones used to solve the Knapsack problem [20]. Of course also less complex solution, like finding the nearest solution as described by White et al. [6] or Felfernig et al. [4], could be sufficient.

VI. USER GUIDANCE – THE BIGGER PICTURE

Indeed, looking at the bigger picture, our vision is to provide such guidance not only to product configuration but also to design modeling and traceability management. We discuss in [12] that the user guidance problem during product configuration is not that different from the user guidance problem elsewhere. For example we want to use technologies developed in this modeling scenario also for modeling with the UML in the context of semantic constraints to ensure different UML views of a system to be coherent [15].

VII. CONCLUSIONS AND FUTURE WORK

In this work, we presented our vision of user guidance for model scenarios, especially product configuration. We described strategies of how to manage and resolve conflicts during the configuration process, which we hope will also be applicable for UML and other modeling scenarios. These strategies are used for resolving conflicts during the configuration process, the reasoning occurs incrementally and is refined with every decision users make.

Once we have evaluated and validated these concrete strategies for the product configuration scenario and demonstrated that they are effective, we are planning to apply our techniques to UML modeling scenarios. During this transition we also plan to refine the roughly outlined strategies for *Trust based on user queries* and *Identifying a Suitable Alternative*. Open issues that also need to be investigated are how to handle several independent conflicts during the configuration process and how these strategies could be applied to a multi-user configurator.

ACKNOWLEDGMENT

This research was funded by the Austrian FWF under agreement P21321-N15.

REFERENCES

- [1] D. Dhungana, R. Rabiser, P. Grünbacher, K. Lehner, and C. Federspiel. *DOPLER: An Adaptable Tool Suite for Product Line Engineering*. in *11th International Software Product Line Conference (SPLC 2007), Proceedings: The Second Volume*. 2007. Kyoto, Japan: Kindai Kagaku Sha Co. Ltd., Tokyo.
- [2] T. Asikainen, T. Männistö, and T. Soininen, *Using a Configurator for Modelling and Configuring Software Product Lines based on Feature Models*, in *Workshop on Software Variability Management for Product Derivation in conjunction with Software Product Line Conference*. 2004: Boston, Massachusetts, USA.
- [3] B. Yu and H.J. Skovgaard, *A Configuration Tool to Increase Product Competitiveness*. IEEE Intelligent Systems, 1998. **13**(4): p. 34-41.
- [4] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker. *Intelligent Support for Interactive Configuration of Mass-Customized Products*. in *Industrial and Engineering Applications of Artificial Intelligence and Expert Systems IEA/AIE*. 2001. Budapest, Hungary.
- [5] P. Trinidad and A. Ruiz-Cortés, *Abductive Reasoning and Automated Analysis of Feature Models: How are they connected?*, in *VaMoS. 2009: Sevilla, Spain*. p. 145-153.
- [6] J. White, D.C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés, *Automated Diagnosis of Product-Line Configuration Errors in Feature Models*, in *Software Product Lines, 12th International Conference*. 2008: Limerick, Ireland. p. 225-234.
- [7] M. Davis, G. Logemann, and D. Loveland, *A machine program for theorem-proving*. Commun. ACM, 1962. **5**(7): p. 394-397.
- [8] R. Balzer. *Tolerating Inconsistency*. in *Proceedings of 13th International Conference on Software Engineering (ICSE)*. 1991.
- [9] S. Fickas, M. Feather, and J. Kramer, *Proceedings of ICSE-97 Workshop on Living with Inconsistency*. . 1997, Boston, USA.
- [10] *DELL Website*. [Accessed February 12th, 2009]; Available from: <http://www.dell.com/>.
- [11] M.L. Rosa, W.M.P.v.d. Aalst, M. Dumas, and A.H.M.t. Hofstede, *Questionnaire-based variability modeling for system configuration*. Software and System Modeling, 2009. **8**(2): p. 251-274.
- [12] A. Nöhner and A. Egyed, *Optimizing User Guidance during Product Configuration*. 2009: unpublished.
- [13] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, *Automated Reasoning on Feature Models*, in *CAiSE*. 2005. p. 491-503.

- [14] A. Egyed. *Instant Consistency Checking for the UML*. in *Proceedings of the International Conference on Software Engineering (ICSE)*. 2006.
- [15] A. Egyed. *Fixing Inconsistencies in UML Design Models*. in *Proceedings of the International Conference on Software Engineering* 2007.
- [16] A. Egyed, E. Letier, and A. Finkelstein. *Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models*. in *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE)*. 2008. L'Aquila, Italy.
- [17] H.v. Maaren and S. Wieringa. *Finding Guaranteed MUSes Fast*. in *11th International Conference, SAT*. 2008. Guangzhou, China.
- [18] C.M. Li and F. Manyà, *MaxSAT, Hard and Soft Constraints*, in *Handbook of Satisfiability*, A. Biere, et al., Editors. 2009, IOS Press.
- [19] R.J. Wallace and E.C. Freuder, *Heuristic Methods for Over-Constrained Constraint Satisfaction Problems*, in *Over-Constrained Systems*, M. Jampel, E.C. Freuder, and M.J. Maher, Editors. 1995, Springer.
- [20] G. Borradaile, B. Heeringa, and G.T. Wilfong, *Approximation Algorithms for Constrained Knapsack Problems*. CoRR, 2009. abs/0910.0777.

Optimizing Non-functional Properties of Software Product Lines by means of Refactorings

Norbert Siegmund, Martin Kuhlemann, Mario Pukall
 Department of Computer Science
 University of Magdeburg
 Magdeburg, Germany
 Email: {nsiegmun,mkuhlema,pukall}@ovgu.de

Sven Apel
 Department of Informatics and Mathematics
 University of Passau
 Passau, Germany
 Email: apel@uni-passau.de

Abstract—Today, software product line engineering concentrates on tailoring the functionality of programs. However, we and others observed an increasing interest in non-functional properties of products. For example, performance, power awareness, maintainability, and resource consumption are important non-functional properties in software development. Current product line techniques have the potential to flexibly optimize non-functional properties. In this paper, we present our vision of optimizing non-functional properties in software product lines. We show how such an optimization can be achieved using refactorings and present first results of a case study.

Index Terms—software product lines; non-functional properties; product derivation;

I. INTRODUCTION

Software product lines (SPLs) are used to generate a variety of related programs that are tailored to specific use cases [1], [2]. By reusing assets in different variants (i.e., programs), SPLs achieve a rapid product deployment and reduce costs. To generate a tailor-made variant, a stakeholder selects the features (functionality) according to her requirements. This way, users can avoid an overhead in functionality for a variant such as a full featured database system in an embedded system. However, tailoring the variant regarding functionality alone is often not sufficient. In practice, *non-functional properties (NFP)* gain momentum. Power awareness, as a non-functional property, is a promising research field [3], [4]. In *Green IT*, alternative implementations of special algorithms such as sorting [5], are developed to reduce power consumption. Non-functional properties are especially important in the field of resource-constrained systems in which binary size and memory consumption are limiting factors. These heterogeneous non-functional requirements often lead to a redevelopment of already existing functionality.

Software product line engineering has been proven to be useful to tailor a variant for functional and non-functional requirements without the negative impact of redeveloping large parts of a software. Variability provided by an SPL should enable the generation of variants that are equal with respect to functionally but differ in their non-functional properties. To this end, SPLs should provide alternative implementations of the same functionality that are optimized for specific NFPs. For instance, by implementing a feature in different ways,

e.g., a performance optimized variant and a footprint optimized variant of a feature. These implementations introduce new variation points in the SPL to be exploited during the configuration process [6], [7].

Our aim is to provide differently optimized variants of an SPL based on a single architecture which is different from other approaches [8], [9]. The positive effect of having a single architecture is that software evolution and maintainability is easier. While the general idea of optimizing NFPs includes also the selection of alternative implementations, in this paper, we focus on refactorings. *Refactorings* are changes in the structure of source code without altering the program semantics [10]. We categorize suitable refactorings according to their influence on non-functional properties in Section III-0a. For example, refactoring *Inline Method* can increase the performance, however, it might also have a negative effect on binary size. Based on our categorization, a user chooses suitable refactorings that optimize the source code during the configuration process. Each refactoring is defined in a single module, called *refactoring feature module (RFM)* [11], and is applied based on the configuration process. This way we can change a variant according its non-functional properties independently of the compiler or programming language, e.g., by decreasing the binary size by selecting the *Pull up Method* refactoring or by increasing the performance through *Method Inlining*. We make the following contributions: (a) We present an overview of tasks that are required to optimize NFP of SPL variants. (b) We show a concrete optimization technique based on refactorings including a proof of concept.

II. VISION

The configuration of an SPL is guided by a feature model. A feature model is created by a domain engineer to define the features of the SPL [12], [13]. Using a feature model as the basis for the variant configuration, it should be possible to optimize NFP of the desired variant. Before a variant can be optimized for an NFP, we have to provide mechanisms that allow a user to measure and configure the property. This is a non-trivial task because the properties are heterogeneous in their nature. For example, even though we can easily measure the binary size of individual features and can aggregate these values for a specific variant, it is difficult to measure Security

and Reliability. The reason is that it is difficult to define a metric for those properties and even harder to find a meaningful aggregation function in order to compare different variants.

There are some models in literature that classify non-functional properties across the software life cycle [14], [15]. Whereas these classifications provide a good overview of possible non-functional properties, they are insufficient for our needs. In order to enable the configuration of such properties, we need a new classification in which each class provides different measurement techniques and configuration mechanism. In [6], we presented three classes of non-functional properties. The first class, called *Directly Assigned Properties*, contains the properties that cannot be quantified. This is the case for Security or Reliability, for which a domain expert cannot define comparable values. These properties can be seen as non-functional features because their configuration completely correspond to the common feature selection. However, not every NFP can be represented as a single non-functional feature, e.g., the footprint of features. The missing quantification affects the optimization and configuration because we have no comparable values, we cannot define optimizations. However, we can hint the user to features that have a positive or negative impact on the required NFP; we directly assign the property to corresponding features in the feature model.

For the category *Inferred Properties*, we can measure the influence of a feature regarding a property, e.g., the influence of a feature on the binary size of a variant. Using different user-defined metrics, we can measure or estimate values for single features and annotate these values to the corresponding feature in the feature model. This allows us to compute in advance the aggregated value for a variant which, for example, can be used to compute the influence of differently selected alternative implementations of a single feature. As a result, a user can define objective functions for desired non-functional properties and an optimizer can then select the best configuration. For example, a user might want to minimize the Power Consumption and keep the footprint below 200 KBytes. Considering V is the set of all valid variants of an SPL and *Footprint* represents the binary size of a variant whereas *Power* the power consumption respectively. Then she could define the following objective function to derive the optimal variant v_{opt} :

$$v_{opt} = v_i \Leftrightarrow v_i(\text{Power}) \leq v_j(\text{Power})$$

$$\wedge v_i(\text{Footprint}) < 200KB \wedge v_{i,j} \in V \wedge i, j \in \mathbb{N} \wedge i \neq j$$

The last category covers *Runtime Properties* which emerge only in a running variant. This makes these NFPs difficult to measure because we first have to configure, compile, and run a candidate variant in order to measure its NFPs. Prominent examples of *Runtime Properties* are Performance and Memory Consumption. Due to the measurement effort, we propose the configuration process to incrementally reduce the number of candidate variants so that only a few variants have to be executed [6].

A. Configuration of Non-functional Properties

The configuration of a variant begins with a user's feature selection. This step defines the functionality a variant has to provide. During the next step, a stakeholder selects the features that improve a non-functional property (category *Directly Assigned Properties*). Although, such a selection is only an extension of the feature selection phase, it is important for the optimization of non-quantifiable properties. A tool can support the stakeholder in this phase by highlighting and grouping suitable features. Thus, the difference between step 1 and 2 is the reason for the feature selection, i.e., a user selects features for purpose of required functionality in step 1 and in step 2 for NFP-optimizing features. During the subsequent step, a user defines constraints regarding non-functional properties, e.g., a variant must not exceed a footprint more than 200 KBytes. These constraints reduce the number of possible acceptable variants, which is important for the measurement of *Runtime Properties*. Afterwards, a user can define an objective function for optimizing a certain property. Based on such a function, the respectively best implementations are automatically selected for runtime measurements.

Whereas in current approaches, the configuration of a variant's functionality constitutes the end of optimization, we apply further optimizations through refactorings. We apply refactorings that have an impact on non-functional properties but do not alter the functionality. These refactorings, defined by a developer or automatically generated during the configuration, can be seen as additional configuration options of the SPL for improving desired NFPs. User defined refactorings can then be selected in step 2. When a refactoring is not part of the SPL but would contribute to the desired NFP, we generate an RFM accordingly. In Section III, we present in detail which refactorings can be automatically applied to improve a certain variant and where manually defined refactorings should be used.

To realize our vision, we have to extend the common SPL development process [13]. In Figure 1, we provide an overview¹ of such an enriched development process. In the upper part, the usual SPL development process is shown. It starts with the domain analysis in which features are identified and the granularity of the variability is specified. Developers continue to implement the defined features. Different techniques are possible which also impact the configuration and optimization of NFPs.

SPLs are often implemented with preprocessor statements like *#IFDEFs* in C and C++ or with components. Also new techniques, e.g., *aspect-oriented programming* [16] and *feature-oriented programming (FOP)* [17], [18] can be used. FOP is a technique to encapsulate feature code in distinct feature modules (FM) (see Figure 1). By selecting features in the product derivation step, the corresponding feature modules are composed to create the desired product. We propose to add a new concurrent process which focus on non-functional

¹Note, that the SPL development is usually separated in domain and application engineering.

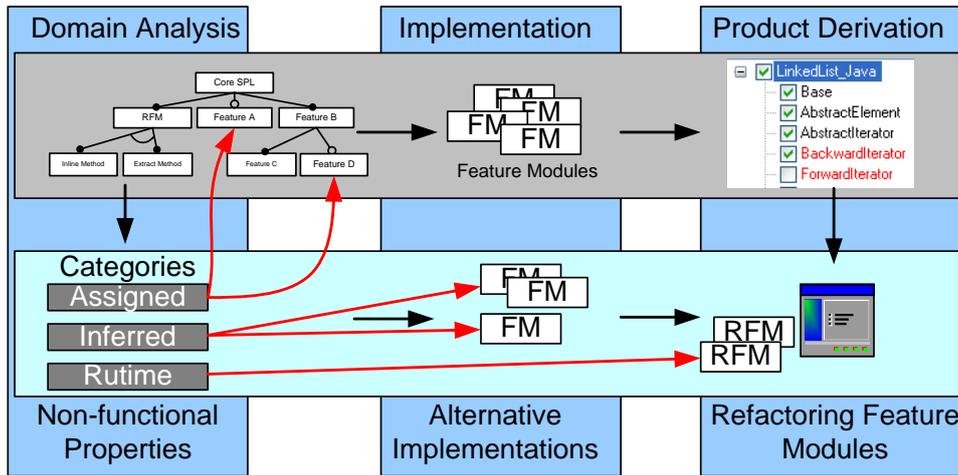


Fig. 1. SPL development including the optimization of non-functional properties.

properties. An additional development team is responsible to identify important properties for the domain, to develop alternative implementations, and to define refactorings (inside RFMs) for these properties. The development of such additional feature models is separated from the common implementation of functionality, e.g., the implementation of alternatives for a new customer with specific needs in a NFP. This way, the development of functionality is independent from the optimization of NFPs. With such an enriched product line engineering approach, we can improve the maintainability of the SPL's source code (separated feature modules for different NFPs). Additionally, we can decrease time-to-market, because one engineering team focuses on the functionality while another team is responsible for tailoring the SPL variants regarding NFP requirements or target systems. As the presented additional process does not affect the common SPL development, existing SPLs can adapt this methodology.

B. Measurement Framework

Due to the large area in which SPLs can be applied, metrics for the same NFP can often not be reused across different SPLs. Commonly, a domain has a strict specification for NFPs and the corresponding metrics. For example, Performance metrics in the area of database management systems are often evaluated with benchmarks whereas the same property, e.g., in SOA, is often expressed in terms of system response time. The metrics and corresponding optimizations, we have already implemented and tested, are specific to the actual domain, but we need a general methodology that allows users to integrate their own specific metrics and aggregation functions.

Based on this insight, we claim that a *framework for measuring and aggregating NFPs* is required in which SPL developers can plug in their domain specific metrics. The framework should provide basic functions that measures individual features (*Inferred Property*) or variants (*Runtime Property*) based on these metrics. Inside a plugin, a developer can use an existing tool (e.g., for measuring the cyclomatic complexity), of a program or feature. The framework could

pass automatically each feature into the plugin which in turn can pass the feature to the desired measurement tool. The results can then be annotated to the respective feature in a feature model. To aggregated the values of different features for a variant the plugin must also define an aggregation function, e.g., for cyclomatic complexity it might be the "maximum". During the configuration, a user can now define a constraint to keep the complexity below a certain number. Using the aggregation function, variants that cannot fulfill this requirement are removed. We use a first implementation of this framework for our refactorings in order to generate refactorings based on user-defined metrics.

We have given an overview how non-functional properties are related to SPLs. We described a classification to highlight that the optimization and configuration of non-functional properties require different methodologies and reflected some optimization possibilities. In the following section, we present a new technique that does not affect the architecture of a variant, i.e., the selected features and feature modules. Therefore, this technique can be used on top of already existing approaches and provides further opportunities to tailor a variant according non-functional properties.

III. OPTIMIZING NFPs WITH REFACTORINGS

Refactorings alter the structure of source code without changing the application behavior [10]. Depending on the type of refactoring, different non-functional properties can be affected. For example, the *Inline Method* refactoring can improve the execution time because it replaces the method call with the body of the called method. A recent study has shown that removing delegation can improve the performance of a program by 50% [19].

Applying refactorings provides new optimization possibilities to a user. We want to exploit these possibilities and select refactorings according to the given optimization goals. Besides such an optimization, a further advantage of this approach is compiler and platform independence because the refactorings are applied to the source code of a variant. Furthermore,

developers are not forced to implement their SPL in a particular way, e.g., coding guidelines defined by a customer can be realized after development. The developers can define the refactorings separately and keep the core architecture of the SPL stable for maintenance. Besides the definition of refactorings by the developers, it should be possible to generate refactorings according to a certain metric. For example, if a user wants to optimize the performance with a given metric for method inlining, a tool should automatically select suitable refactorings or generate them if they do not already exist in the SPL.

Given these requirements, we developed a technique to define and reuse refactorings in SPLs like feature modules. *Refactoring feature modules (RFMs)* integrate refactorings with feature-oriented programming (FOP) [11]. The goal of both techniques, RFM and FOP, is to successively transform a base program. Whereas modules in FOP transform the functionality of the base program, RFMs transform the structure of the base program. Once defined or generated, RFMs become user-selectable features in a feature model.

The main focus of RFMs so far was in program integration [11]. RFMs are defined by the programmer as part of the product line design and selected by the user in order to overcome incompatible structure of a variant with an external application. For instance, to reuse an existing library in a client application, classes of this library might need to be renamed in order to be compatible [11].

We use RFMs to manipulate non-functional properties. While RFMs can be defined and selected manually for integration purposes, their manual definition and selection becomes unfeasible when they should adapt NFPs. The reason is that, to improve the NFP Performance, potentially hundreds or thousands of RFMs must be defined and selected, of which each inlines one method (*Inline Method* refactoring).

a) *Selection of Refactorings*: There are numbers of refactorings described in literature. Our study builds on the refactorings defined by Martin Fowler [10]. In a first step, we analyzed the refactorings and came up with an approximated influence on NFPs for every refactoring. The analysis based on the known influence of different program executions when applying different refactorings, e.g., removing setter methods can result in less compiler instructions and thus may improve the performance. We plan to evaluate in a detailed case study the influence of the most common refactorings. The results are given in Table 1. Note, that applying a refactoring *can* improve or degrade a property but does not have to. We need additional metrics, e.g., those for method inlinings used in compilers, to achieve the desired effect. In the following, we describe the results of our analysis exemplary for some NFP:

- **Performance.** To reduce the execution time for method calls a programmer can apply refactorings like *Inline Method*, *Inline Class*, *Remove Middleman*. This is done by replacing a call with the called method's body. The method call is removed but the same actions happen as before, so performance is improved. However, when methods grow too large, this results in cache mismatches

of the processor [20]. These mismatches arise because the method is too large to fit in the cache completely and instead must be reloaded, which increases the execution time. To overcome such problems different metrics exist, e.g., for compilers to achieve the best performance.

- **Footprint.** The footprint of an application is the sum of the footprint of each compiled file. For Java, we measure the class files that contain intermediate byte code. By removing (setting) methods or code clones (e.g., by transforming members to parent classes using *Pull up Field* or *Pull up Method* refactorings), the footprint can be reduced. Note, that these refactorings often have only a small influence to shrink the binary size. In contrast, for example inlining methods in multiple other methods may result in an expanded footprint. This must be considered when footprint constraints are defined.
- **Coding styles.** Coding styles are important if products are sold as source code libraries to multiple customers where each customer has its own styling guideline. There are different tools that check the validity of code against coding rules, e.g., Checkstyle². For such rules, refactorings (e.g., *Extract Method* or *Rename Method*) can be automatically generated and applied on demand. Although, this approach might be a possible way to pass a program validator, the maintainability for developers will rather be decreased because of generated names for methods and variables. A possible solution are developer-defined RFMs that are selected on demand. With RFMs functional requirements of an SPL are separated from non-functional requirements, e.g., different code guidelines of different customers. A variant can be quickly adapted to fit the needs of new customers when existing RFMs are reused.

We have collected *some* possible use cases for optimizations using refactorings. The suitability of each refactoring depends on the program and on the quality of the metric that defines which refactorings have to be used. Both types of usage, generation and manual definitions are required. Automatically generated refactorings are useful if a high number of refactorings is necessary. Manually implemented RFMs should be used if developers knowledge is necessary and soft properties like "Readability" must be improved.

IV. PROOF OF CONCEPT

Currently, we are developing a tool which tackles the following requirements: (a) configuration of a variant regarding functionality and *Directly Assigned Properties*, (b) definition of objective functions for NFPs, (c) automated selection of optimal implementations regarding NFPs, and (d) the definition, selection, and appliance of RFMs. The NFP optimizer supports SPLs implemented with feature-oriented programming (FeatureC++ [21] and JAK [18]). In [6], we have shown a possible solution to compute an optimal selection of alternative

²<http://checkstyle.sourceforge.net/>

Non-functional property	Improve	Decrease
Performance	Inline Method, Inline Class, Remove Middleman, Remove Setting Method, Replace Delegation with Inheritance, Replace Temp with Query, Inline Temp	Encapsulate Field, Extract Class, Extract Method, Form Template, Introduce Assertion Method, Hide Delegate, Replace Inheritance with Delegation, Self Encapsulate Field, Change Unidirectional to Bidirectional, Decompose Conditional
Footprint	Collapse Hierarchy, Pull up Constructor Body, Pull up Field, Pull up Method, Remove Middleman, Remove Setting Method	Decompose Conditional, Encapsulate Field, Extract Class, Extract Interface, Hide Delegate, Inline class, Inline Method, Inline Temp, Introduce Assertion, Introduce Explaining Variable, Push down Field, Push down Method, Remove Assignments to Parameters, Self Encapsulate Field
Styling Guidelines and Code Metrics	Extract Method, Replace Conditional with Polymorphism, Replace nested Conditional with Guard Clauses, Extract Method, Create Template Method, Consolidate Duplicate Conditional Fragments	Inline Method, Replace Exception with Test, Inline Method
Readability	Extract Class, Extract Subclass, Extract Superclass, Inline Method	Collapse Hierarchy, Consolidate Conditional Expression, Decompose Conditional, Encapsulate Field, Extract Method, Hide Delegate, Inline Class
Object Size	Inline Temp	

TABLE I
OVERVIEW OF REFACTORINGS AND THEIR INFLUENCE ON NON-FUNCTIONAL PROPERTIES.

implementations for the NFPs Cyclomatic Complexity, Footprint, and Performance without using refactorings. We extend the tool to support our approach by using RFMs. After the derivation of a variant, the user has now the opportunity to further improve a certain non-functional property. Currently, we only support Performance, however, in future we will provide optimizations for additional NFPs.

After a variant is configured, we use JastAdd³ to analyze the abstract syntax tree of the composed variant. In particular, we search for methods where refactorings can be applied in order to improve performance. We additionally analyze where inlinings might be reasonable, e.g., method calls in loops. However, we exclude methods that are polymorphic and recursive, as inlining those methods is not yet implemented. The output of this analysis is a list of candidate methods (see left part of Figure 2).

Such a listing is interesting for stakeholders with programming skills because methods can be manually marked for inlining where a positive effect can be foreseen. This is the same as the *inline* keyword in C or C++.⁴ However, normally a user defines a metric that defines when methods should be inlined, see top right in Figure 2. This enables a compiler-independent definition of an inlining metric and has therefore effects independent of the underlying system. After defining refactorings beneficial for NFPs of SPL variants, we compute and generate (or reuse existing) RFMs to reach the optimization goals. The selected refactorings are shown to the user (right part of Figure 2). Subsequently, the tool applies the RFMs and compiles the new variant. The performance of the synthesized product is measured and compared to the performance of the product without refactorings. Although, we currently support only inline method refactorings, the results in Figure 3 show that we achieve performance improvements

for certain cases (when we applied RFMs, we never produced an inferior result compared to the non-optimized variant).

b) Case Study: We present our first results for the optimization of NFP using RFMs. For our case study, we used the micro benchmark presented in [19], which implements a delegation chain.

After the analysis, the system came up with 980 possible methods for inlining. We defined a metric to restrict the method size to 1000 statements and the maximal inline depth to 10. Applying more sophisticated metrics that cover more setup possibilities, e.g., preferred inlining in loops, is left for future work. Our tool computed 120 Inline Method refactorings based on our metric. As no RFMs were present in the SPL before, our tool generated all of them. After applying these RFMs, we measured the performance 10.000 times to get significant data. The results are given in Figure 3. The X-axis shows the intervals for time needed to pass the performance test. For each interval, we counted the number of executions which are depicted in the Y-axis. In the result, the execution times were significantly better with our optimizations than without. We found that refactorings can be successfully applied to an already optimized variant to further improve an NFP. The refactorings have also an influence on the footprint property. In the unoptimized version the sum of all class files requires 705,569 bytes whereas the variant optimized using refactorings consumes 833,021 bytes. We see our assumptions according the influence of refactorings to NFPs approved and expect additional optimization benefits if more refactorings are supported.

V. RELATED WORK

Product derivation tools try to guide the user through the whole derivation process. There are commercial tools like pure::variants [22] and Gears [23] that contain mechanisms to maintain and develop an SPL as well as scientific tools [24], [25], [26]. These tools allow developers to create feature models and guide users through the configuration process with

³<http://jastadd.org>

⁴The *inline* keyword is used in front of method declarations to force the compiler to inline the method.

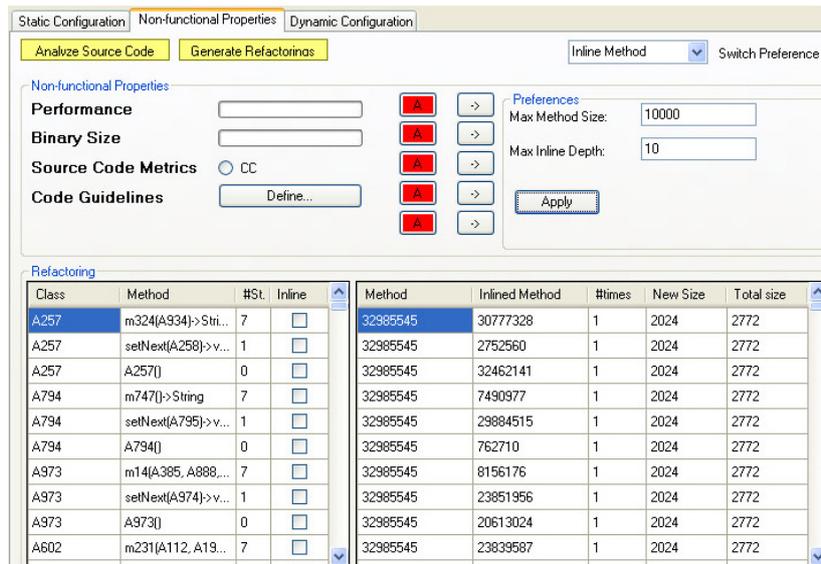


Fig. 2. Generating and applying refactorings in the NFP optimizer tool.

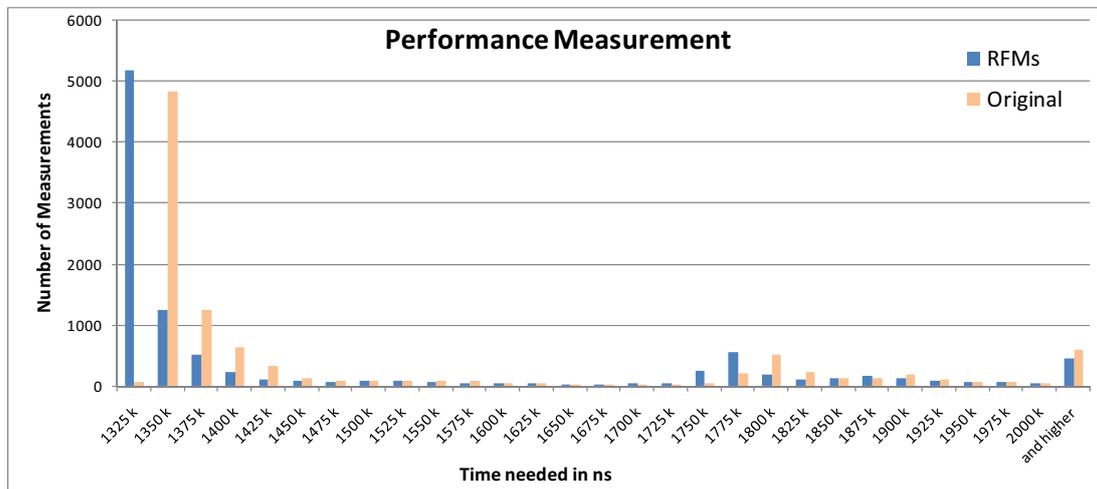


Fig. 3. Results for measuring a variant with and without RFMs.

special visualization techniques. Neither the measurement of non-functional properties nor the optimization for NFPs is supported for SPLs.

Benavides et al. [27], [28] presented a technique based on Constraint Satisfaction Problems (CSP) solvers to seek an optimal variant. The solver evaluates values attached to features in the feature model and then computes an optimal configuration for a small number of features. White et al. [7], [29] extended this approach to resolve resource constraints in the variant selection process. For large scale problems they propose a Filtered Cartesian Flattening to approximate a good variant. We see both approaches promising for an integration, e.g., for selecting optimal feature modules. However, we further provide an optimization technique based on RFMs and a framework to measure the values needed for optimization.

Other approaches use model-driven engineering techniques

to generate different architectures optimized for certain quality attributes. In [8] components can be differently connected and interfaces are generated to obtain a valid program with modified quality attributes. Kim et al. [9] propose a framework, called DRAMA, which captures the requirements of users. Based on these requirements, different architecture styles, e.g., Layers or Model View Controller, can be applied to improve or degrade a NFP. The framework can also compare alternative implementations and chose the one with the best quality attributes. This approach is similar to our configuration of optimal implementations. We additionally include user selectable refactorings on source code level to further optimize a variant for NFPs and the measurement of NFPs.

Smith [30] uses correctness-preserving transformations in his tool *Kids* to improve the performance of a program. These transformations are similar to refactorings. Unlike Smith's

transformations, RFMs can be seamlessly integrated into the SPL development process because RFMs represent reusable modules that can be described like features. Critchlow et al. [31] present an approach to use refactorings to change the architecture in order to improve certain quality attributes. They consider refactorings not at source code level but at architecture level to flexibly change the components architecture of a variant. We focus on refactorings that are applied to the source code and after a variant is created.

The Skoll project [32] targets on testing and measuring applications with large configuration spaces. The project tries to overcome the problem of having a huge amount of products by using a large number of users that share their computation power. For measuring runtime properties, this might be a suitable approach. However, the effort is very high and it will not scale with a large SPL. Our refactoring feature modules can be applied independent from the variant space.

Zhang et al. propose to use Bayesian Belief Network in order to analyze and predict non-functional properties based on the experience of the development of similar products and domain experts [33], [34]. The knowledge is captured and used for the development of new products to achieve suitable decisions for optimizing a certain non-functional property. This approach targets on architectural design and decisions during the design phase. Our approach can be applied after an SPL is developed and is therefore independent of architectural decisions.

VI. CONCLUSION

We presented our vision of optimizing non-functional properties (NFPs) of variants of software product lines (SPLs). We outlined difficulties in measuring and configuring NFPs and motivated the need for a framework that allows users to plugin their own metrics. Based on the measured values, our tool selects alternative implementations to optimize certain NFPs. We presented a new approach for optimization based on refactoring feature modules (RFMs). RFMs, defined by developers or automatically generated, are part of the SPL and can be selected like features to further improve NFPs. First results based on automatically generated *Method Inline* refactorings show that performance improvements can be achieved. In future work, we will support additional refactorings to further increase performance. In addition, we will support the optimization of other NFPs. Our long term goal is to provide a framework for which developers can implement plugins that optimize NFPs.

ACKNOWLEDGMENT

Norbert Siegmund is funded by the German Ministry of Education and Science (BMBF), project 01IM08003C. Mario Pukall is funded by German Research Foundation (DFG), project SA 465/31-2. Apels work is supported in part by DFG project #AP 206/2-1. The presented work is part of the

ViERforES⁵, RAMSES⁶, and FeatureFoundation⁷ projects.

REFERENCES

- [1] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
- [2] C. W. Krueger, "New methods in software product line development," in *Proceedings of the International Software Product Line Conference (SPLC)*. IEEE Computer Society, 2006, pp. 95–102.
- [3] R. Jain, D. Molnar, and Z. Ramzan, "Towards understanding algorithmic factors affecting energy consumption: Switching complexity, randomness, and preliminary experiments," in *Proceedings of the Workshop on Foundations of Mobile Computing*. ACM Press, 2005, pp. 70–79.
- [4] C. Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour, "Choosing the "best" sorting algorithm for optimal energy consumption," in *Proceedings of the International Conference on Software and Data Technologies (ICSOFT)*, 2009, pp. 199–206.
- [5] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis, "JouleSort: A balanced energy-efficiency benchmark," in *Proceedings of the 2007 International Conference on Management of Data*. ACM Press, 2007, pp. 365–376.
- [6] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, and G. Saake, "Measuring non-functional properties in software product lines for product derivation," in *Proceedings of the 15th International Asia-Pacific Software Engineering Conference (APSEC)*. IEEE Computer Society, 2008, pp. 187–194.
- [7] J. White, D. C. Schmidt, E. Wuchner, and A. Nechypurenko, "Automating product-line variant selection for mobile devices," in *Proceedings of the International Software Product Line Conference (SPLC)*. IEEE Computer Society, 2007, pp. 129–140.
- [8] P. O. Rossel, D. Perovich, and M. C. Bastarrica, "Reuse of architectural knowledge in SPL development," in *Proceedings of the 11th International Conference on Software Reuse (ICSR)*. Springer-Verlag, 2009, pp. 191–200.
- [9] J. Kim, S. Park, and V. Sugumaran, "Drama: A framework for domain requirements analysis and modeling architectures in software product lines," *Journal of Systems and Software*, vol. 81, no. 1, pp. 37 – 55, 2008.
- [10] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [11] M. Kuhlemann, D. Batory, and S. Apel, "Refactoring feature modules," in *Proceedings of the International Conference on Software Reuse*, 2009, pp. 106–115.
- [12] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.
- [13] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [14] J. A. Mccall, P. K. Richards, and G. F. Walters, "Factors in software quality. Volume I. Concepts and definitions of software quality." General Electric CO Sunnyvale California, Technical Report ADA049014, November 1977.
- [15] "Software engineering - Product quality, Part 1: Quality model," 2001.
- [16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science, vol. 1241. Springer Verlag, 1997, pp. 220–242.
- [17] C. Prehofer, "Feature-oriented programming: A fresh look at objects," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science, vol. 1241. Springer Verlag, 1997, pp. 419–443.
- [18] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Transactions on Software Engineering (TSE)*, vol. 30, no. 6, pp. 355–371, 2004.
- [19] S. Götz and M. Pukall, "On performance of delegation in Java," in *Proceedings of the International Workshop on Hot Topics in Software Upgrades*. ACM Press, 2009, pp. 1–6.

⁵<http://vierfores.de>

⁶http://www.witi.cs.uni-magdeburg.de/iti_db/forschung/ramses/

⁷<http://www.fosd.de/ff>

- [20] J. Dean and C. Chambers, "Towards better inlining decisions using inlining trials," in *Proceedings of the ACM conference on LISP and functional programming*. ACM, 1994, pp. 273–282.
- [21] S. Apel, T. Leich, M. Rosenmüller, and G. Saake, "FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming," in *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, ser. Lecture Notes in Computer Science, vol. 3676. Springer Verlag, Sep. 2005, pp. 125–140.
- [22] D. Beuche, H. Papajewski, and W. Schröder-Preikschat, "Variability Management with Feature Models," *Science of Computer Programming*, vol. 53, no. 3, pp. 333–352, 2004.
- [23] Big Lever, "Gears," <http://www.biglever.com>.
- [24] D. Streitferdt, M. Riebisch, and I. Philippow, "Details of formalized relations in feature models using OCL," in *International Conference on Engineering of Computer-Based Systems (ECBS)*. IEEE Computer Society, 2003, pp. 297–304.
- [25] G. Botterweck, D. Nestor, A. Preußner, C. Cawley, and S. Thiel, "Towards supporting feature configuration by interactive visualization," in *Proceedings of Workshop on Visualisation in Software Product Line Engineering*, 2007, pp. 125–131.
- [26] E. Cirilo, U. Kulesza, and C. P. de Lucena, "A product derivation tool based on model-driven techniques and annotations," *Journal of Universal Computer Science*, vol. 14, no. 8, pp. 1344–1367, 2008.
- [27] D. Benavides, A. Ruiz-Cortés, and P. Trinidad, "Automated reasoning on feature models," in *Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, ser. Lecture Notes in Computer Science, vol. 3520. Springer Verlag, 2005, pp. 491–503.
- [28] D. Benavides, S. Segura, P. Trinidad, and A. R. Cortés, "FAMA: Tooling a Framework for the Automated Analysis of Feature Models," in *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2007, pp. 129–134.
- [29] J. White, B. Dougherty, and D. C. Schmidt, "Selecting highly optimal architectural feature sets with filtered cartesian flattening," *Journal of Systems and Software*, vol. 82, no. 8, pp. 1268–1284, 2009.
- [30] D. R. Smith, "Kids: A semiautomatic program development system," *IEEE Trans. Softw. Eng.*, vol. 16, no. 9, pp. 1024–1043, 1990.
- [31] M. Critchlow, K. Dodd, J. Chou, and A. van der Hoek, "Refactoring product line architectures," in *International Workshop on Refactoring: Achievements, Challenges, and Effects*, 2003, pp. 23–26.
- [32] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan, "Skoll: Distributed continuous quality assurance," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2004, pp. 459–468.
- [33] H. Zhang, S. Jarzabek, and B. Yang, "Quality prediction and assessment for product lines," in *Advanced Information Systems Engineering (CAiSE)*. Springer, 2003, pp. 681–695.
- [34] H. Zhang and S. Jarzabek, "A bayesian network approach to rational architectural design," *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 4, pp. 695–718, 2005.

Automating the Configuration of Multi Software Product Lines

Marko Rosenmüller, Norbert Siegmund
 School of Computer Science
 University of Magdeburg, Germany
 {rosenmue,nsiegmun}@ovgu.de

Abstract—The increased use of software product lines (SPLs) results in complex software systems in which products of multiple SPLs communicate and interact with each other. Such a system of interdependent SPLs has to be considered as a whole to achieve compatibility between different SPL instances. In this paper, we present an approach to design and configure *multi software product lines (MPLs)*, i.e., product lines that consist of multiple interdependent SPLs. Therefore we use *composition models* that describe how an MPL is composed from multiple SPL instances. This allows us to automate the configuration of MPLs which is required to handle the resulting complexity. We also show how to automatically derive configuration generators to further simplify the configuration process and we report from experiences of applying the presented approach.

I. INTRODUCTION

Software product lines (SPLs) enable reuse by generating software from a common set of assets, e.g., by composing components [5]. The *instances* of an SPL, i.e., the products, can be programs, libraries, and also components. Hence, products of an SPL can also be used for building more complex SPLs [17]. For example, a component can be developed as an SPL and can be combined with other components in a larger SPL. Due to the success of SPLs, more and more programs are developed as product lines and integrated in complex systems. This results in *product lines of product lines* or *nested product lines* [11]. We call arbitrary compositions of SPLs *multi software product lines (MPLs)*.

As an example for an MPL, consider a sensor network (SNW) that consists of network nodes (SNW-Nodes) which are small embedded devices. The software running on network nodes differs in functionality because the nodes have to accomplish different tasks: there are *sensor nodes* for sensing data, *access nodes* that provide access to the sensor network, and *data nodes* that aggregate and store data [12], [14]. Developing an SPL for network node software, allows a user to generate tailor-made variants for the different node types. The software of a single network node consists of multiple programs and libraries. For example, a data management system might be used for data storage, cryptographic libraries ensure confidentiality of data, and communication libraries might be used to provide basic communication functionality. These programs or libraries are more and more developed as SPLs to increase reuse and to minimize the functional overhead especially needed in the domain of embedded systems. This results in SPLs that *use* other SPLs for their realization.

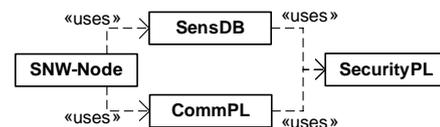


Fig. 1. Uses relationships between different product lines.

An example for this *uses* relationship between SPLs in a sensor network is shown in Figure 1. The SNW-Node SPL uses an SPL for data storage (SensDB) and a communication product line (CommPL). Product lines CommPL and SensDB encrypt transferred and stored data using the security product line (SecurityPL) which provides cryptographic algorithms. Hence, there are four interdependent SPLs and each of them has to be configured according to the requirements of the other SPLs.

In contrast to a single SPL, we have to consider the functional dependencies between the different instances of all involved SPLs. That is, modifying the configuration of one instance might require a different configuration of other SPL instances. Already manual configuration of single SPLs is highly complex and error prone. Manual configuration of large networks of interdependent SPLs might be impossible if many features and dependencies between features are involved. Furthermore, the configuration process has to be repeated when the configuration or the implementation of one SPL changes. Ideally, a user only has to configure one SPL that describes the whole application scenario, e.g., a sensor network product line, and does not have to care about implementation details of underlying SPLs. A solution could be to integrate multiple feature models into a single feature model. This, however, results in a large feature model that mixes different domains and provides configuration options that are not important for the problem domain.

In this paper, we present an approach to automate the configuration of MPLs based on *composition models* as described in [13]. A composition model integrates multiple SPLs by describing for each SPL which instances of other SPLs it *uses*. It thus describes dependencies between concrete SPL instances. In order to integrate composition modeling in the SPL engineering process, we show how a composition model can be semi-automatically derived from the domain models

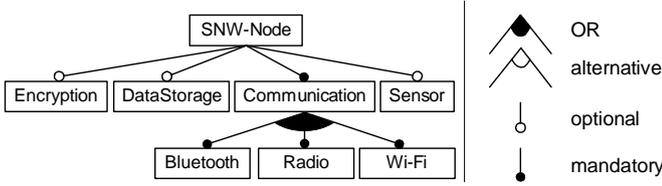


Fig. 2. Feature diagram of a sensor network node (SNW-Node) SPL.

of the SPLs of an MPL. Finally, we present an approach to generate configuration generators for MPLs, i.e., programs that are used to derive configurations for all SPL instances of an MPL.

II. MODELING MPLS

While SPL engineering is well understood and programs can be automatically generated from SPLs, the configuration of multiple interdependent SPLs (i.e., MPLs) is mostly not considered. In the following, we describe how the feature modeling approach can be used for modeling MPLs. However, we will show that feature models do not provide a suitable solution for describing dependencies in arbitrary MPLs, e.g., when multiple instances of the same SPL are used in an MPL. Furthermore, using only feature models for MPL modeling results in complex solutions. For that reason we introduce composition models and show how they overcome existing problems.

A. Feature Models

An SPL is used to create similar programs that share a common set of *features*. The features of an SPL are distinguishable characteristics that are of interest to some stakeholder [5]. SPLs can be described using *feature models* that are often visualized using *feature diagrams* [9], [5]. An example for a sensor network node (SNW-Node) product line is depicted in Figure 2. A concrete program or *instance* of an SPL is defined by a selection of required features. *Domain constraints* of a feature model are used to ensure only valid feature combinations in an SPL configuration. For example, *requires* and *mutual-exclusion* relations are used to describe dependencies between features [5]. In general, arbitrary propositional formulas might be used [2].

Domain constraints can not only be used to describe dependencies within a single SPL but also between different SPLs [6]. For example, a *requires* constraint

$$SnwNode.Bluetooth \Rightarrow CommPL.Bluetooth \quad (1)$$

describes that when a user selects feature BLUETOOTH of the SNW-Node SPL ($SnwNode.Bluetooth$; cf. Fig. 2) also feature BLUETOOTH of the communication framework SPL ($CommPL.Bluetooth$; cf. Figure 1) has to be selected. However, if multiple instances of the same SPL are required, there can also be constraints between SPLs that cannot be described on the domain level. For example, communication between nodes in a sensor network, requires support for the same

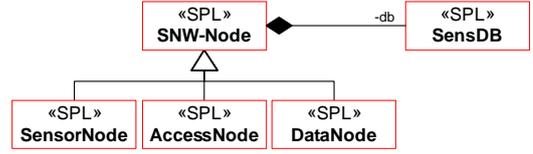


Fig. 3. Composition model with an SPL for sensor network nodes (SNW-Node) and specialized SPLs SensorNode, AccessNode, and DataNode. The SNW-Node SPL uses an instance of a sensor database (SensDB).

communication protocol in the nodes (e.g., RADIO in Fig. 2). That is, two instances of the same SPL (e.g., one instance for sensor nodes and one instance for access nodes) have to be configured to support the same protocol. A domain constraint cannot describe this dependency because SPL instances cannot be distinguished in the domain model.

A similar problem occurs when an SPL A uses two different instances b_1 and b_2 of SPL B . Since the domain model does not include SPL instances, it is hard to use domain constraints to describe such dependencies. For example, to describe that feature f_3 of instance b_2 has to be selected whenever feature f_1 of SPL A is selected, we might use a listing of the distinguishing features of b_2 as part of the constraint:

$$A.f_1 \wedge (B.f_1 \wedge B.f_2) \Rightarrow B.f_3 \quad (2)$$

where $(B.f_1 \wedge B.f_2)$ describes instance b_2 . At first glance, this correctly represents the required constraint; however, it is also valid for other instances of B that include features f_1 and f_2 which is not intended. A direct representation of SPL instances could avoid such problems and simplify the configuration at the same time. Furthermore, a constraint as shown in Equation 2 is not part of the problem domain of SPL A . Actually, it is an implementation issue of A and for a different scenario SPL B might be replaced by a different SPL. Hence, this implementation knowledge should be separated from the domain model of A .

B. Composition Models

In order to overcome the presented problems, we introduced *composition models* that describe an MPL by modeling a composition of multiple SPL instances [13]. This allows us to describe the dependencies of all SPLs and to provide means for automatically configuring the SPLs according to these dependencies. In the following, we review composition models and show how these are used to describe MPLs.

SPL Instances: A composition model uses the concept of aggregation of classes known from OOP to represent the uses relationships between SPLs. Each class represents an SPL and a class instance represents an instance of an SPL. In Figure 3, we depict the UML representation of a composition model for the sensor network example. SNW-Node and SensDB are SPLs and the aggregation relation between them denotes that SNW-Node uses an instance of SensDB with name `db`.

Specialization: In product line engineering, *specialized SPLs* are used to describe a subset of the variants provided by an SPL [6]. We include SPL specialization into composition

models using inheritance between SPL classes. This allows us to easily reuse SPL configurations in different MPLs. In Figure 3, *SensorNode*, *AccessNode*, and *DataNode* are specializations of the *SNW-Node* SPL. For example, *SensorNode* is a partial configuration of *SNW-Node* in which feature *SENSOR* is mandatory.

Constraints: In order to achieve compatibility between SPL instances of an MPL, we use composition model constraints. A constraint in a composition model is a propositional formula¹ consisting of features similar to a feature model constraint [2]. A feature in that expression can be referenced using the name of an SPL or the name of an SPL instance. When using an SPL name, it refers to the feature in all instances of that SPL (i.e., a usual domain constraint between feature models) and when using the name of an SPL instance, it refers only to a feature in that concrete instance which we call *instance constraint*. The constraint shown in Equation (2) thus simplifies to $A.f_1 \Rightarrow b_2.f_3$, where A denotes an SPL and b_2 denotes an SPL instance. If needed, instances can be fully qualified with the name of the SPL an instance belongs to. For example, $A.b_2$ refers to instance b_2 defined in SPL A . Constraints might also include specialized SPLs. For the sensor network example in Figure 3, we can specify constraints:

$$\text{SensorNode.Radio} \Rightarrow \text{AccessNode.Radio} \quad (3)$$

$$\text{SnwNode.Encryption} \Rightarrow \text{SnwNode.db.Encryption}. \quad (4)$$

Constraint (3) describes a dependency between *SensorNode* and *AccessNode* specializations of the *SNW-Node* SPL. Selecting feature *RADIO* in *SensorNode* requires to select feature *RADIO* in all instances of *AccessNode* to ensure a compatible communication protocol between sensors and access nodes. In contrast, instance constraint (4) addresses only the *SensDB* instance db defined in *SNW-Node*. Hence, different instances of *SensDB* can be configured differently, e.g., with or without encryption, depending on the kind of the node.

Constraints are part of an SPL and are inherited when creating a specialized SPL. For example, SPL *SensoreNode* in Figure 3 inherits all constraints stored in *SNW-Node*, e.g., constraint (4). Constraints can also be redefined in a specialized SPL. However, since a specialization represents a subset of the variants represented by its parent, a constraint can only be redefined by adding propositions using a conjunction. That is, constraint redefinitions can only reduce the number of possible variants.

Conditional Dependencies: Sometimes an SPL instance of an MPL is only required when some optional feature is available in a configuration. For example, the instance of *SensDB* in Figure 3 is only needed when feature *DATAS-*

¹First order logic might also be used, e.g., to provide constraints for sets of SPL instances. However, we think that propositional logic might be sufficient because a limited number of SPL instances should be the usual case and thus quantification is not required. Nevertheless, when using more complex constraints in domain models [7] the same kind of constraints should be used for composition models.

TORAGE of *SNW-Node* is selected (cf. Fig. 2). To describe this, we use *conditional dependencies* which define optional SPL instances that are only needed when a particular feature or a set of features is present in a configuration [13]. This simplifies the configuration process because the SPL only has to be configured when the according feature is selected.

III. AUTOMATING THE CONFIGURATION OF MPLS

In order to automate MPL configuration and to integrate MPL modeling into the development process for SPLs, we extend the product line engineering process with composition modeling which is part of the domain design process. A composition model connects domain model and implementation of an MPL by describing the SPL instances used to implement an MPL. As illustrated in Figure 4, the following steps are required for creating MPLs:

- creating a feature model for an MPL,
- generating and refining composition models,
- deriving a configuration generator.

In the following, we describe the required steps in detail and discuss experiences when creating the example sensor network MPL in Section IV.

A. Feature Models for MPLs

We describe an MPL using the feature models of all contained SPLs and a composition model that defines how the SPL instances are combined. Depending on the SPLs, we can differentiate between *hierarchical* and *flat* MPLs.

Hierarchical MPLs: The dependencies between the SPL instances of an MPL often result in a *hierarchy of SPLs*. In Figure 3, *SNW-Node* uses an instance of *SensDB* and thus defines a hierarchy between both SPLs. The dependency between the SPLs is directed: *SNW-Node* depends on *SensDB* but *SensDB* is independent of *SNW-Node*. An SPL hierarchy may have multiple levels, e.g., as shown for *SNW-Node* in Figure 1, resulting in a three level hierarchy. We call *SNW-Node* the *top-level SPL* of the hierarchy because there is no other SPL that depends on *SNW-Node*. Usually, we can describe the variability of a whole MPL using the feature model of the top-level SPL.

Flat MPLs: There can also be MPLs that do not exhibit a hierarchy which we call *flat* MPLs. Examples are MPLs of communicating programs such as in a client-server architecture where client and server are developed as distinct SPLs and have to be configured to achieve compatibility. For example, a mail client and a mail server have to support the same communication protocol, e.g., IMAP. Flat MPLs also occur when multiple instances of the same SPL are combined in an MPL. For example, an MPL of replicated DBMS stores data in a master DBMS and in a slave for replication. Such a system can be developed as a single SPL from which differently configured instances for master and slave can be generated. The MPL thus consists of multiple instances of the same SPL. Usually, the dependencies between the SPLs of a flat MPL are not directed but each of the SPLs depends on the other. In contrast to hierarchical SPLs, there is no top-level

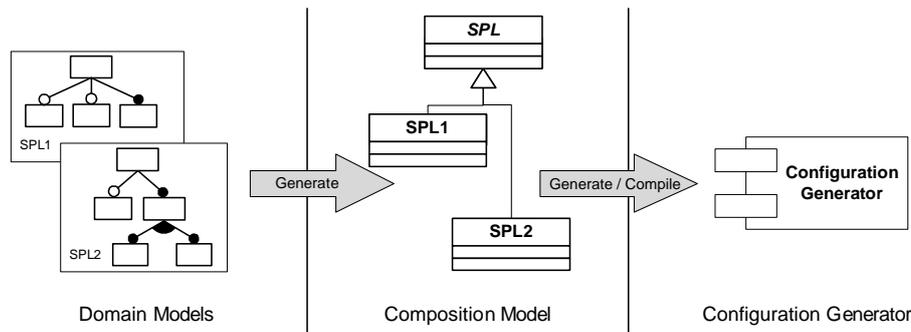


Fig. 4. Generating composition models and configuration generators for MPLs.

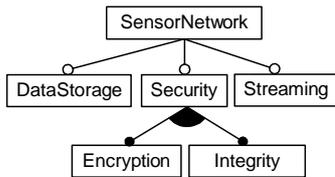


Fig. 5. Feature diagram of the SensorNetwork MPL.

feature model that describes the MPL. Imposing a hierarchy by defining one or the other SPL as the top-level SPL is usually not sufficient because one SPL does not describe the whole MPL and variability of the other SPL is hidden. For example, the feature model of a mail server is inappropriate for configuring the whole MPL, i.e., mail server and mail client. A better solution is to introduce a hierarchy by creating a new top-level feature model for the MPL. This way, we can describe the variability of the whole MPL using only a single feature model while variability of the constituent SPLs is hidden, e.g., a feature model for a replicated DBMS that describes the whole MPL and abstracts from underlying SPL instances for master DBMS and slave for replication.

In the sensor network example, a product line of sensor network nodes (cf. Fig. 1) is a hierarchical MPL since it uses multiple other SPLs. In a complete sensor network scenario, however, there are different specializations of network nodes used (SensorNode, AccessNode, DataNode) which results in a flat *sensor network MPL*. We thus create a new top-level feature model that represents the whole sensor network, as depicted in Figure 5. It describes which functionality the sensor network provides and abstracts from the underlying specializations of the SNW-Node SPL. We use features like `DATA STORAGE` to represent an optional data node of the sensor network which is implemented by the specialized DataNode SPL. Which features are to be included in an MPL feature model we discuss in Section IV.

B. Creating Composition Models

Based on the feature models of an MPL, the composition models can be created in a step-wise manner. In fact, this will be the usual scenario because a developer of an SPL will create a composition model for her SPL that can be

reused in higher level SPLs and hides underlying SPLs. Each composition model defines the directly used SPL instances and is independent of higher level SPLs which is important for reusing the model. For example, we can easily replace SensDB (cf. Fig. 3) with a different DBMS product line. This requires to store the composition model of each SPL separately. The composition models of different SPLs are implicitly connected via the uses-relationships between SPLs defined as instance variables (e.g., instance `db` in Figure 3). Hence, the compound composition model of an MPL is the union of all composition models of the underlying SPLs. Replacing an SPL thus only requires to modify the instance variable that defines the type of an SPL instance, e.g., modify the `db` instance of SNW-Node to be of a type other than SensDB.

As shown in Figure 4, we use a composition model generator to create initial composition models and their UML representation. The model can be created at any time in the design process and can be updated when SPLs change or different specialized SPLs become available. For generating composition models, we use the integration of C# and UML in Microsoft Visual Studio 2008. The generator creates a C# class for every SPL and an inheritance relation to represent SPL specialization. The use of the partial class concept of C# and the integration with UML class diagrams allows an SPL developer to edit the composition model using either the UML representation or the C# code while the generated corresponding representation is automatically updated. The classes of the composition model are subclasses of an abstract SPL class, as shown in Figure 6. This abstract class implements generic functionality of the configuration generator as we describe later.

A composition model can be visualized using a UML diagram which is also used by the SPL developer to define SPL instances. An integrated visualization of a complete MPL composition model can be generated by including all SPL classes in a single UML diagram, as shown for the sensor network example in Figure 7. Such an integrated view of the whole model is usually not necessary, since it is sufficient to edit the models of all SPLs separately.

SPL Instances and Constraints: The initially generated composition model is refined by an SPL developer who creates instance variables to represent the uses-relationships between

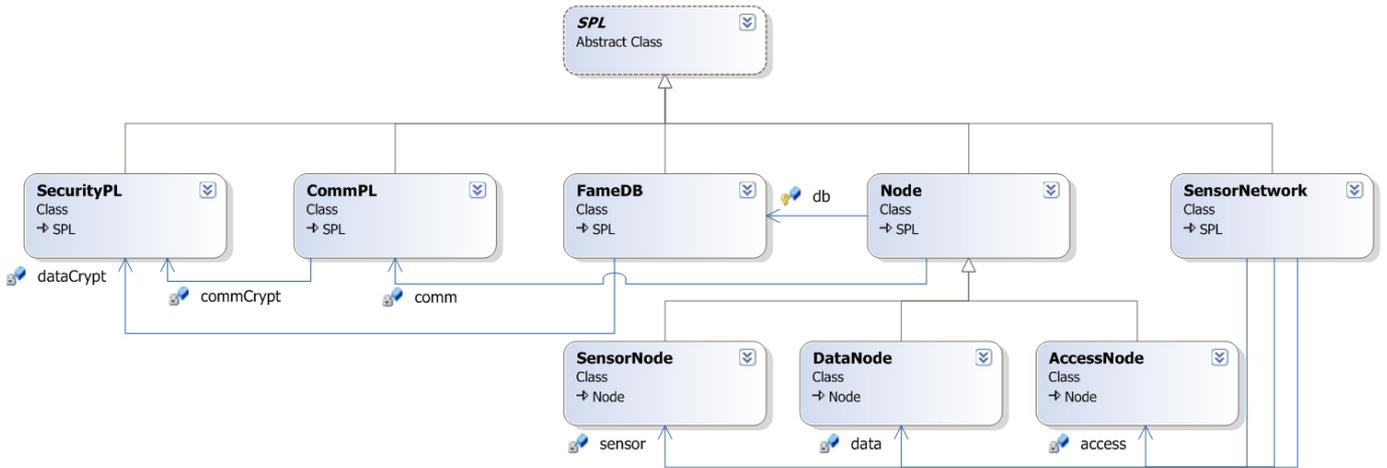


Fig. 7. Screenshot of the compound composition model for the SensorNetwork MPL.

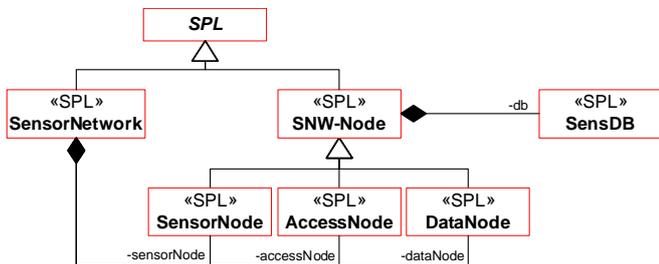


Fig. 6. A composition model for a SensorNetwork SPL that uses a network node SPL (SNW-Node).

SPLs. The SPL instance variables connect individual composition models. For example, instance variable `sensorNode` of the SensorNetwork SPL in Figure 6 represents an instance of the specialized SPL `SensorNode` and connects both SPLs. SPL instances can be added via the UML representation of a composition model or by directly changing the source code of the SPL classes.

A domain modeler creates composition model constraints to define which features from used SPL instances are required by a higher level SPL depending on its configuration. As discussed above, constraints are arbitrary propositional formulas between features of the involved SPLs but may refer only to concrete SPL instances if required. For example, we can define constraints that refer only to the `dataCrypt` instance of `FameDB` and do not affect the `commCrypt` instance of `CommPL` (cf. Fig. 7). Finally, conditional dependencies are created to represent optional SPLs that are only needed when a particular feature is selected. For example, a data node is only needed when a user selects the `DATA STORAGE` feature of the SensorNetwork SPL.

C. Configuration Generators for MPLs

Having an MPL, described by feature models of all SPLs and an integrated composition model, we can automatically derive a configuration generator. This configuration generator

is used in an interactive configuration process and asks a user for configuration decisions (i.e., feature selections) required to configure all SPL instances of an MPL and checks for violation of constraints.²

We implemented a generic configuration generator using C#. The configuration generator provides a graphical user interface that asks a user for feature selections of all required SPL instances for an MPL scenario. It checks for violations of model constraints and creates the SPL instances required for an MPL configuration. Parts of the generic implementation are provided by abstract class `SPL`, as shown in Figure 4. The generic implementation of the configuration generator is extended by MPL specific code which is the composition model of the MPL (subclasses of `SPL` in Fig. 4 and 7). Each MPL specific class extends the functionality of the abstract SPL class by defining instance variables for the used SPL instances. The configuration generator code (generic code plus MPL specific composition model) is compiled into an executable program.

IV. EXPERIENCE REPORT AND DISCUSSION

In the following, we shortly report about our experience of modeling and configuring MPLs using the sensor network example. Since we observed different possible ways for creating feature models for MPLs and defining constraints, we discuss benefits of the respective solutions.

A. Handling MPL Variability

For the sensor network example, we defined specialized SPLs of the SNW-Node SPL (`SensorNode`, `DataNode`, and `AccessNode`), as shown in Figure 6. Since it is a *flat* MPL, we created a feature model that represents the whole MPL as already described (cf. Fig. 5). We think that such an additional feature model for flat MPLs is often useful for several reasons:

²Currently, the configuration generator does not support redefinition of model constraints in specialized SPLs.

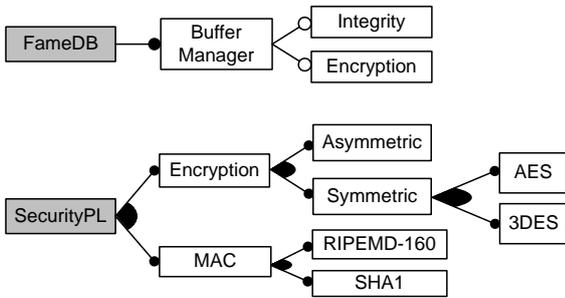


Fig. 8. Excerpts of the feature diagrams of FameDB and SecurityPL.

- It provides a simple structure for an MPL and simplifies the configuration process because only a single feature model is needed. Moreover, some of the features of the SPLs are not important for configuring the MPL, e.g., because they are mandatory in the particular MPL.
- It reduces configuration options, e.g., by creating features that imply a set of features of lower level feature models.
- It avoids invalid configurations, e.g., it implicitly defines constraints that otherwise had to be expressed as propositional formulas.
- It allows developers to define new features representing features of lower level feature models. For example, it is simpler for a domain expert not familiar with cryptography to configure a feature ENCRYPTION instead of a feature AES that represents the encryption algorithm.

An MPL feature model is useful to reduce the configuration space to valid combinations of SPL instances and to provide only configurations that are meaningful in a special context, e.g., to simplify testing and maintenance of an MPL by supporting only a predefined subdomain. However, restricting the variability using an MPL feature model also means that features important for a particular application scenario might not be available in the MPL's feature model. In order to be able to create configurations also for such scenarios a user can additionally configure the underlying SPLs.

For some MPLs, it might be useful to avoid an additional feature model, e.g. when most of the features of the lower level feature models have to be repeated in the MPL's feature model. Including all features in a single feature model is usually not a solution because it results in large feature models that are probably unmanageable. Furthermore, it is a question of additional effort for creating and maintaining the additional feature model.

B. Composition Model and Constraints

Based on the feature models, we generated four composition models for SPLs CommPL, FameDB, Node, and SensorNetwork consisting of eight classes representing SPLs or specialized SPLs (cf. Fig. 7). We extended the composition model to define SPL instances and constraints. SPL instances can be defined by editing each model separately but also using an integrated view, as shown in Figure 7.

We use several constraints to ensure valid configurations of the sensor network SPLs. For example, the FameDB SPL requires only AES and SHA1 algorithms of the SecurityPL (cf. Fig 8) and does not use all provided algorithms. We describe this with *instance constraints* between SPLs FameDB and SecurityPL:

$$FameDb.Encryption \Rightarrow FameDb.dataCrypt.AES \quad (5)$$

$$FameDb.Integrity \Rightarrow FameDb.dataCrypt.SHA1 \quad (6)$$

Constraint (5) defines that the AES encryption algorithm of the SecurityPL is required when feature ENCRYPTION is selected in FameDB. Constraint (6) defines that the SHA1 hash algorithm is required when feature INTEGRITY is selected in FameDB. Instance constraints are stored as part of the composition model of the according SPL which simplifies their reuse. For example, constraints (5) and (6) are stored as part of FameDB and can be reused in other compositions of FameDB.

Instance Constraints vs. SPL Specialization: As an alternative solution to instance constraints, we can create specializations of an SPL. For example, we can create a specialization of SecurityPL that is used for data encryption and integrity only in FameDB. This specialization can already include features AES and SHA1 thus avoiding instance constraints.

In contrast to instance constraints, SPL specialization provides better means for reuse in other MPLs. The reason is that instance constraints are defined per instance variable (e.g., `FameDb.dataCrypt`) and cannot easily be reused. On the contrary, specialized variants can be reused simply by defining the corresponding instance variables. However, specialization does not scale because different application scenarios with different constraints result in an exponentially growing number of specializations. Hence, there is a tradeoff between simpler reuse of specializations and an increasing effort for defining and managing an increasing number of specializations. In general, it seems to be more favorable to create a specialized SPL when it can be used multiple times and includes many configuration decisions. Constraints are to prefer when reuse is limited and many specializations with few design decisions can be avoided. In many cases, this probably results in a mixture of specialization and instance constraints as we observed it for the sensor network.

C. The Configuration Process

Based on domain and composition models, we automatically derive a configuration generator and use it to configure all SPLs used in an MPL. The configuration process starts with the top-level feature model and an empty feature selection. In our example, this is the SensorNetwork feature model. Based on mandatory features, the configuration generator creates a configuration for each instance variable and continues recursively. For example, SensorNode and AccessNode are always required in a sensor network and thus a `sensor` and `access` configuration is created according to the composition model in Figure 7. Initially there is no configuration for SPL DataNode because a conditional dependency defines that instance `data` of SensorNetwork is only available when

feature `DATA_STORAGE` is selected. This configuration is only added when the feature is selected by the user. Similarly, selecting feature `ENCRYPTION` adds an instance of `SecurityPL` to SPLs `FameDB` and `CommPL`.

When the user has provided a configuration for all SensorNetwork features, the configuration of the `SensorNetwork` SPL is finished. This should be the usual scenario, but a detailed configuration process of lower-level SPLs might be needed. For example, the `FameDB` SPL can be manually configured with features that influence the performance of the DBMS, like special index data structures. Such additional configurations are also required if configuration decisions are missing. Each time a user changes a configuration, the configuration generator checks the validity of the whole composition.³

D. Avoiding Code Duplication

For some application scenarios it is important to avoid code duplication of SPL instances that are used multiple times in an MPL. For example, `SensDB` and `CommPL` use an instance of `SecurityPL` for encrypting and decrypting data (cf. Fig. 7). In a scenario without product lines, e.g., using a library for security algorithms, we could reuse this library for data storage and communication, e.g., to reduce the binary size or used working memory of the resulting compound system. Reusing product lines in the same way is not always possible, because the required configurations of the instances might be contradicting. For example, when two alternative features of `SecurityPL` are needed (cf. Fig. 7) two different instances of the same SPL are required, one for `SensDB` and one for `CommPL`.

In order to optimize an MPL configuration with respect to code reuse, the configuration generator detects possible SPL instances that can be reused. However, it cannot always be decided automatically whether an instance can be reused or not. For example, even when reusing an SPL instance with more features than needed, the semantics of the reused SPL might be different, e.g., it might be required to initialize additional features. Furthermore, due to the feature interaction problem [4], the behavior of one feature might change when adding another feature. Finally, reuse is problematic if it means to reuse a running program or component including its internal state. In order to avoid such errors, a user can select whether an SPL can be reused or not. However, to provide a practical solution, there should be more sophisticated configuration mechanisms included in the future, e.g., using insights from component based software development to solve issues regarding the internal state of an SPL instance.

V. RELATED WORK

As described in Section II, approaches to model SPLs can also be applied to model MPLs. Czarnecki et al. use cardinality-based feature models with constraints to specify specializations and constraints in feature models where multiple selections of one feature are possible [6], [10]. The used *feature model references* and *feature cloning* can also

be applied to model compositions of SPL instances as needed for MPLs. The tool *FeaturePlugin* furthermore allows for configuring multiple feature models that are integrated via feature model references [1]. Including multiple feature models into a single MPL feature model via feature model references results in highly complex domain models for large MPLs. Furthermore, it mixes domain modeling with SPL implementation because lower level SPLs are used for *implementing* higher level SPLs and the domains are only related due to the implementation. An approach that integrates feature models of different hardware and software product lines was presented by Streitferdt et al. [15]. The presented integration of multiple product lines does not consider SPL instances or constraints between them which is not needed in their context.

In contrast to the modeling approaches presented above, we propose to model SPL instances and dependencies between them. Our approach is an extension of existing SPL modeling techniques and we think that their combination is required to sufficiently model MPLs. We think that domain constraints are required for describing dependencies between the SPLs of an MPL but implementation issues like constraints to lower level SPLs, used for implementation of higher level SPLs, should be handled separately, i.e., in composition models.

Czarnecki et al. showed that feature models provide less descriptive power than ontologies but are easier to use due to their specialization [7]. The relationship of composition models to ontologies is similar: Compositions of SPL instances can also be modeled using ontologies and a composition model is a special view representing dependencies between related SPLs. Hence, feature models and composition models are special techniques with a focus on certain aspects of SPL modeling. Ontologies might be used to integrate both for a more complete description of an MPL.

Product populations built from Koala components are described by van Ommering [17]. Koala components can be built by composing smaller components at configuration time which is different from our work. We aim at describing how SPLs have to be composed to build a larger product line, i.e., an MPL, and not concrete products, i.e., Koala components. With our approach, the composition of a complex product (e.g., a component), built from other products, automatically changes depending on a feature selection, which is a modification of the composed architecture. This is different from manual composition of components to derive a larger component.

Fries et al. present an approach to model SPL compositions for embedded systems [8]. They use *feature configurations*, a selection of features, to describe a group of instances that share these features. Feature configurations are similar to specialized SPLs in staged configuration but do not allow a user to describe multiple configuration steps. A composition model, as described in [8], is defined for a complete composition of product line instances. We create a composition model for each SPL of an MPL and integrate them to describe MPLs which eases reuse of composition models. Implementation issues or reusing SPL instances are not addressed in [8].

SPLs consuming different other SPLs in a SOA environment

³Currently, we detect violation of constraints and do not check satisfiability.

are described by Trujillo et al. [16]. Their focus was on modeling the interfacing between SPLs in a service-oriented environment. This includes service registration and service consumption.

Tools like *pure::variants*⁴ and *Gears*⁵ allow a domain engineer to build feature models and describe dependencies among them. Both tools support modeling dependencies between SPLs and Gears explicitly supports nested product lines that can be reused between different feature models. *guidsl* is a tool to specify composition constraints for feature models using a grammar [2]. It provides means to check models and interactively derive a configuration for a feature model. All these tools support model checking using constraints similar to our approach. However, they do not consider different instances of an SPL within one composition and *guidsl* does not consider the integration of multiple SPLs at all.

Batory et al. have shown that SPL development using layered designs scales to *product lines of program families* [3]. The focus of their work is on generating families of programs from a single code base and reasoning about program families. The work does not address composition of SPLs developed independently or compositions of SPL instances.

VI. SUMMARY

With increasing importance of SPLs, techniques to model, develop, and configure compositions of multiple interacting SPLs have to be provided. In this paper, we presented an approach to model and configure such *multi software product lines* (MPLs) that are built from multiple interdependent SPLs. The presented work is a first step to extend current SPL engineering with a new process that describes the implementation of MPLs on an abstract level using a composition model of involved SPL instances. We thus bridge the gap between domain modeling and implementation of MPLs using a high-level abstraction.

We have shown that composition models for MPLs can be generated which reduces the effort for modeling MPLs and integrates composition modeling into product line engineering. Based on domain and composition models of an MPL, we can automatically create configuration generators that help to automate the configuration process of MPLs. In order to reuse composition knowledge, we use a separate composition model for each SPL that can be easily used in different MPLs. The composition model of a whole MPL is derived by combining the composition models of all constituent SPLs.

As a next step, we want to evaluate the approach using existing product lines and analyze how the configuration process of large MPLs can be further simplified. For example, by checking satisfiability we could provide early feedback to the user that configures an MPL.

ACKNOWLEDGMENT

The work of Marko Rosenmüller is funded by German Research Foundation (DFG), project number SA 465/34-1.

Norbert Siegmund is funded by German Ministry of Education and Research (BMBF), project number 01IM08003C. This work is part of the projects MultiPLe⁶ and ViERforES⁷.

REFERENCES

- [1] M. Antkiewicz and K. Czarnecki, "FeaturePlugin: Feature Modeling Plug-in for Eclipse," in *Eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*. ACM Press, 2004, pp. 67–72.
- [2] D. Batory, "Feature Models, Grammars, and Propositional Formulas," in *Proceedings of the International Software Product Line Conference (SPLC)*, ser. Lecture Notes in Computer Science, vol. 3714. Springer Verlag, 2005, pp. 7–20.
- [3] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin, "Generating Product-Lines of Product-Families," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society Press, 2002, pp. 81–92.
- [4] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature Interaction: A Critical Review and Considered Forecast," *Comput. Netw.*, vol. 41, no. 1, pp. 115–141, 2003.
- [5] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [6] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Staged Configuration Using Feature Models," in *Proceedings of the International Software Product Line Conference (SPLC)*, ser. Lecture Notes in Computer Science, vol. 3154. Springer Verlag, 2004, pp. 266–283.
- [7] K. Czarnecki, C. H. P. Kim, and K. T. Kalleberg, "Feature Models are Views on Ontologies," in *Proceedings of the International Software Product Line Conference (SPLC)*. IEEE Computer Society Press, 2006, pp. 41–51.
- [8] W. Friess, J. Sincero, and W. Schroeder-Preikschat, "Modelling Compositions of Modular Embedded Software Product Lines," in *Proceedings of the 25th Conference on IASTED International Multi-Conference*. ACTA Press, 2007, pp. 224–228.
- [9] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.
- [10] C. H. P. Kim and K. Czarnecki, "Synchronizing Cardinality-Based Feature Models and Their Specializations," in *European Conference on Model Driven Architecture Foundations and Applications (ECMDA)*, 2005, pp. 331–348.
- [11] C. W. Krueger, "New Methods in Software Product Line Development," in *Proceedings of the International Software Product Line Conference (SPLC)*. IEEE Computer Society Press, 2006, pp. 95–102.
- [12] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 122–173, 2005.
- [13] M. Rosenmüller, N. Siegmund, C. Kästner, and S. S. ur Rahman, "Modeling Dependent Software Product Lines," in *GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, no. MIP-0802. Department of Informatics and Mathematics, University of Passau, Oct. 2008, pp. 13–18.
- [14] M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, and G. Saake, "FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems," in *EDBT'08 Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, Mar. 2008, pp. 1–6.
- [15] D. Streitferdt, P. Sochos, C. Heller, and I. Philippow, "Configuring Embedded System Families Using Feature Models," in *Proceedings of Net.ObjectDays*. Gesellschaft für Informatik, 2005, pp. 339–350.
- [16] S. Trujillo, C. Kästner, and S. Apel, "Product Lines that Supply Other Product Lines: A Service-Oriented Approach," in *SPLC Workshop: Service-Oriented Architectures and Product Lines - What is the Connection?*, Sep. 2007.
- [17] R. van Ommering, "Building Product Populations with Software Components," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM Press, 2002, pp. 255–265.

⁴<http://www.pure-systems.com>

⁵<http://www.biglever.com>

⁶http://www.witi.cs.uni-magdeburg.de/iti_db/research/MultiPLe

⁷<http://www.vierfores.de>

Variability in Time — Product Line Variability and Evolution Revisited

Christoph Elsner^{*}, Goetz Botterweck[†], Daniel Lohmann[‡], Wolfgang Schröder-Preikschat[‡]

^{*} Siemens Corporate Technology & Research, Erlangen, Germany

christoph.elsner.ext@siemens.com

[†] Lero – The Irish Software Engineering Research Centre, Limerick, Ireland

goetz.botterweck@lero.ie

[‡] Friedrich-Alexander University Erlangen-Nuremberg, Erlangen, Germany

{wosch,lohmann}@cs.fau.de

Abstract—In its basic form, a variability model describes the variations among similar artifacts from a structural point of view. It does not capture any information about *when* these variations occur or *how* they are related to each other in time. This abstraction becomes problematic as soon as time-related aspects become essential for the modeling purpose, e.g., when providing long-term support for a product line or when planning its future strategy.

In this paper, we provide an overview of approaches that deal with time-related aspects in variability, which is summarized under “variability in time”. In contrast, the modeling of structural commonalities and differences is referred to as “variability in space”. We take an inductive approach and survey different uses of the term “variability in time”, which turn out to be orthogonal. We generalize the uses and identify three different types: variability of linear change over time (maintenance/evolution), multiple versions at a point in time (configuration management), and binding over time (product derivation). We validate the types by using them to describe complex product line evolution scenarios where they exhibit expressive and discriminatory power. Finally, we go into depth for the first type (maintenance/evolution) and identify the tasks and aspects to be considered when building a detailed evolution research agenda in the future.

I. INTRODUCTION

Variability can be seen as “the ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context.” [30]. Similarly, variability modeling techniques “aim at representing the variability in a product family”. [27] Well-known techniques are feature modeling [14], orthogonal variability modeling [22], COVAMOF [28], and decision modeling [2].

These techniques, however, describe the variability of a product line for one particular moment in time only. Given the inevitable change of a software system over time, several authors added the time dimension to variability. Whereas the classical notion of variability is referred to as “variability in space” the new dimension is called “variability in time”¹ [22], [9], [24], [32], [16], [19], [4]. For ease of reading, when we use the term variability without any qualification in the following, we actually refer to “variability in space”.

¹We regard the terms “variability in time” and “variability over time”, both in singular or plural, as synonyms.

The topic of “variability in time” is of considerable importance and has also been discussed in projects and workshops ([21], [33], [15]). However, recent uses and definitions of the term refer to widely differing time-related variability issues. Authors claiming to address “variability in time”—according to their definition—therefore actually only address a *subset* of time-related problems. Whereas there exists substantial work and a common understanding on what “variability in space” is about, this is not the case for “variability in time”. For time-related variability issues in product line engineering, an established body of knowledge, which allows for decomposing and structuring the problem in an appropriate way, is still missing. This paper performs first steps towards such a common body of knowledge.

We take an inductive approach and survey different uses of “variability in time” (Section II). They turn out to be orthogonal, so we generalize them and identify three types of variability in time: variability of linear change over time (maintenance/evolution), multiple versions at a point in time (configuration management), and binding over time (product derivation) (Section III). This gives an expressive terminology for characterizing complex time-related variability interrelations, which we illustrate for several product line evolution scenarios (Section IV). Finally, we go into depth for the first type (maintenance/evolution), for which we define tasks and aspects (future planning, present modeling and tracking, past analysis) (Section V), serving as a prerequisite for defining a research agenda in the future.

II. VARIABILITY IN TIME IN THE LITERATURE

In this section we report on the uses of the term “variability in time” in recent work. Interestingly, it only appears in papers within the product line context. The term has been used to describe three different problem areas: product line evolution, product line versioning, and product line binding variability.

A. Product Line Evolution Variability

One definition recited several times [19], [12], [34] is of Pohl et al. [22]: “Variability in time is the existence of different versions of an artefact that are valid at different times.” This definition is not without problems. Having different versions

of an artifact is obviously not specific to software product lines. It holds for every evolving software system. However, even for single system development, it is common that one product is released in different versions and, thus, one has to deal with different artifact versions at the *same* time. This definition, however, implicates that there is a straight flow of artifact versions, each new version actually invalidating its predecessor.

Other authors [24], [7] use the term “variability in time” to describe not only the change of the artifact versions over time, but also of their *variability dependencies* (denoting the “variability in space”) over time. As requirements change over time, the product line must evolve as well. For a product line this means adding, removing, or changing features, as well as adding, removing, or changing *variability dependencies* (e.g., mandatory, optional, alternative). Still, however, they do not address multiple artifact versions valid at the *same* time.

B. Product Line Versioning Variability

In [32], in turn, the problem of “variability in time” is seen as a problem of dealing with multiple valid versions at the current moment in time. Maintaining one product in different versions in parallel is already a challenge for single product development, as each product consists of a multitude of artifacts of specific versions. For product lines this problem is even more difficult; a multitude of products, each possibly having several released versions, must be related to the correct artifact version for maintenance.

C. Product Line Binding Variability

Other authors, finally, use the term “variability in time” to describe the creation time and binding time of variability [4], [6], [12], [13]. During domain engineering, at certain phases of system development (e.g., analysis, design, coding compilation, linking, distribution, installation, start-up, or run-time) variation points are “made explicit”, meaning foreseen, designed, or implemented in a dedicated way. In application engineering, these variation points are successively bound to specific variants, decreasing the variability until at the end one specific system is the end product.

III. ASSIGNING THE DIFFERENT NOTIONS TO TYPES

As can be seen from the previous section, the notion of the term “variability in time” varies considerably, each adopting a different viewpoint. However, the uses actually are orthogonal. Thus, in this section, we assign the different notions of the term to types, which will help to reason about product line variability over time providing a reasonable terminology. We generalize the three above notions and subordinate more time-related variability research, both from general software engineering and product line research, to which we will give exemplary literature references. We distinguish between the following three types of variability in time: variability of change over time (maintenance/evolution), multiple versions at a moment in time (configuration management), and binding over time (product derivation).

A. Variability of Linear Change over Time: Maintenance and Evolution

Meta-studies indicate that 50 to 90 percent of costs refer not to development of software products but to their changing [17]. As described in Section II, the popular definition of “variability in time” in [22] denoting the existence of different artifact versions at different times does not address this issue appropriately in the context of software product lines.

The discipline of software engineering providing comprehensive concepts, methods, and tools for changes is called *software maintenance* or *software evolution*. For this paper, we will not strictly distinguish between both terms. However, commonly, the term maintenance has a rather “reactive” connotation (i.e., corrective or adaptive maintenance according to [18]), whereas evolution tends to be used in a more “proactive” way (i.e., perfective or preventive maintenance in [18]). In the following we will use the more appropriate term, respectively.

Evolving a product line is much more complex as in the case of single systems, since the variability (in space) changes over time as well. Formalizing the different types of product line changes to ensure consistency is a big challenge specific to product line engineering. A taxonomy for the different types of product line evolution operators with a special focus on requirements level changes is given in [26], for example. A more empirical approach categorizing the findings of two case studies into an approach relating the reasons of product line changes to their architectural impact can be found in [29]. Finally, a framework for decentralized evolution of product line feature models is presented in [23]. It facilitates specifying explicitly which change operations are permitted on a feature model by defining allowed deviations with respect to a reference feature model.

B. Variability of Multiple Versions at a Moment in Time: Configuration Management

From a simplified point of view, software maintenance may be seen as a continuous flow of versioning over time, each new version invalidating its predecessor. In this form, it does not concern multiple versions of the same artifact at the same time. This is a task of configuration management.

Software configuration management (SCM) is often defined as a holistic discipline comprising tools, techniques, and processes for tracking and controlling everything related to versioning and changing of software-related artifacts (e.g., [20]). Version management tools such as SVN or ClearCase constitute the technical side of SCM. In this paper, we will use the term configuration management in a narrower sense: managing different versions of the same artifact throughout the software lifecycle, or, as stated informally in [20]: “eliminating the confusion and error brought about by the existence of different versions of artifacts”. Thus, it deals with maintaining the integrity of products considering that they may be comprised of artifacts of different versions.

Once a product is delivered to a customer, keeping it in sync with the product line core assets is often not part of the contract or even feasible (e.g., due to hardware changes).

	Linear change over time	Multiple versions at a moment in time	Binding over time
Focus	Change as continuous flow	Managing artifact versions in parallel	Binding VPs as process in time
Off focus	Multiple artifact versions	Evolving artifacts	Evolution and versioning of VPs
Software engineering field	Maintenance/Evolution	Configuration management	Product derivation
Exemplary PL challenge	Consistent evolution of ViS	Consolidating versions and ViS	Time flexibility for binding ViS

PL: product line | VPs: variation points | ViS: variability in space

TABLE I
OVERVIEW OF THE THREE TYPES OF “VARIABILITY OVER TIME”.

However, the released products have to be maintained as well, for instance by bug-fixing (corrective maintenance) or for possible future update requests (perfective maintenance). Therefore, the configuration of the specific product (i.e., all its artifacts in their specific version) must on the one hand be frozen in time. On the other hand, the product must keep in contact with the evolving core asset base to support the product’s maintenance.

Consolidating versioning and software product lines variability is an important research challenge of product line engineering. It has both been addressed in research prototypes by integrating product line functionality into version management tools (e.g., [31]) or vice versa (e.g., [32]).

1) *Versions over Time and Continuity*: According to Aoyama [1] evolution can be *continuous or discontinuous*. Continuous evolution corresponds to a set of requirement changes that is small enough, so that only little architectural reengineering is required. Above a certain threshold of changes, it is necessary to re-engineer the architecture, leading to substantial architectural and implementational differences. When the continuity is interrupted, a new product line generation arises and, for a certain period of time, both product lines have to be maintained in parallel. Major architectural reengineering leads to differences that are difficult to track. Dealing with those problems is a further challenge for configuration management.

C. Variability of Binding in Time: Product Derivation

A considerable amount of variability of a product line is planned early in the software product line lifecycle, during scoping [25]. The actual implementation of the variability (foreseeing and implementing variation points) can be performed at different moments in domain engineering, as well as the binding of these variation points to variants for product derivation in application engineering [4]. Various binding techniques ranging from coding time to run-time may be applied.

Although the term “variability in time” has been used in the context of variation point creation and binding, one might argue that it is a rather trivial observation that this is usually done in a process over time. However, we believe that it is possible to gain some remarkable insights if seen in a broader scope. Creation and binding of variability successively over time is also known as multi-level and staged configuration, respectively [8]. Such an approach is required, if it is neces-

sary to model so-called “software ecosystems”, this is when software product lines themselves pass through several distinct organizations until the variability is completely bound [3].

A further variability aspect is introduced if the variation points are designed in a way that it is possible to bind them at different stages (e.g., either at compile time or at run-time). Then, additional variability, so-called “timeline variability” [10] is introduced, leading to an even more complex product configuration over time.

D. Intermediate Summary

Table I summarizes the three aforementioned types of “variability in time”. They differ in what is in and off their focus and in the fields of classical software engineering that address them. To indicate that “variability in time” has specific product-line-related challenges, we also mention one exemplary challenge tackled by recent product line research. Providing a complete list of all relevant challenges is beyond the scope of this paper.

IV. SCENARIOS

In the following, we apply the three different types of “variability in time” to describe several complex product line scenarios where they exhibit expressive and discriminatory power, giving evidence that they provide the right abstraction for talking about variability over time.

Scenario 1: Interaction of Product Line and Products.

Figure 1 shows a product line scenario containing each of the types of “variability in time”. The first type—linear change over time (maintenance)—develops in parallel with the time flow (horizontal axis). The vertical axis, in turn, corresponds to the binding of variability. Interaction of the upper product line evolution and the lower product maintenance corresponds to configuration management, which is the third type. It becomes necessary, for example, when a product bug-fix is fed back into the product line core assets (bottom-up), or, vice versa, a product receives a feature upgrade from the core asset base (top-down).

The figure may be interpreted as follows: For the version of the product line in 2009, binding of variability (the derivation of the product A and B) is done in a single step. The product line evolves and, in 2010, product C (V1) is derived. This product has to be maintained in parallel to the evolving core asset base. After creation of a maintenance release in 2011 C (V2), the adaptations are merged back into the product line, which is a configuration management task. Product D is bound

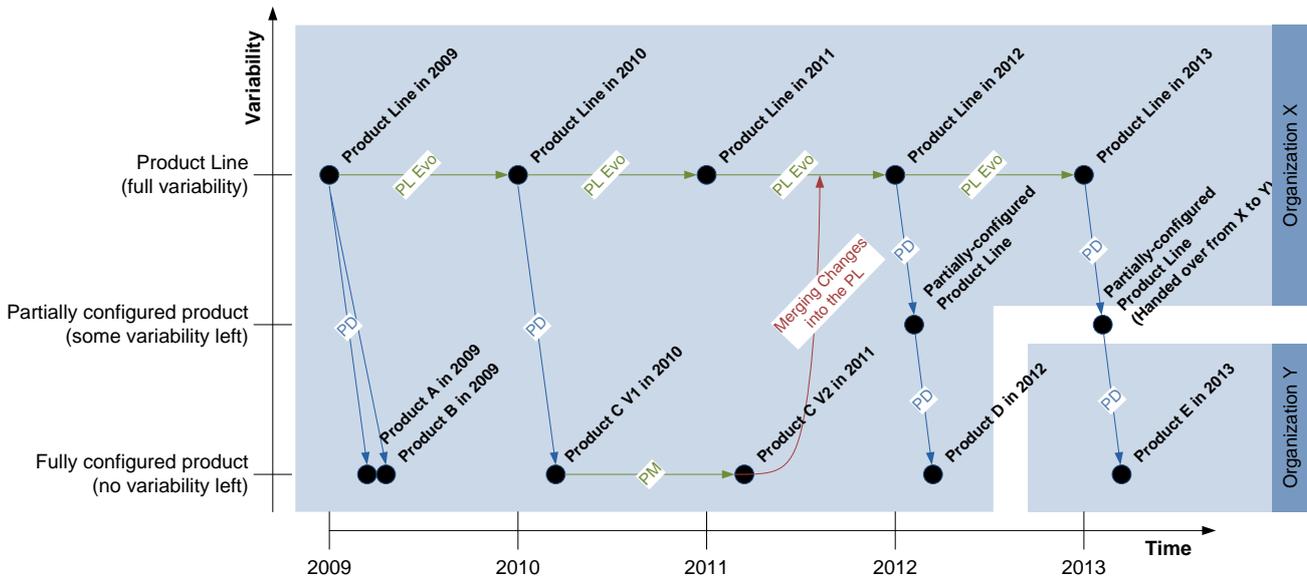


Fig. 1. Product line evolution scenario.

in separate steps (staged configuration [8]). Finally, in 2013, product E results from staged configuration involving multiple parties. The supplier (organization X) hands over the partially-configured product line to its customer (organization Y), which binds the remaining variation points and completes product derivation. This is also referred to as *software ecosystems* [3].

Scenario 2: Interaction of Product Lines. The following scenario (Figure 2) illustrates continuity and discontinuity of product lines. Again, maintenance corresponds to the horizontal axis, configuration management, here not between products and their product line, but between distinct product lines, to the vertical axis. Product derivation is not shown in the figure. Subsequently, we distinguish between product line generations, releases, and revisions using version numbering ($\langle \text{gen} \rangle . \langle \text{rel} \rangle . \langle \text{rev} \rangle$). This terminology is similarly used in other publications (e.g., [29], [1]) and can be found likewise in various software projects.

Starting in 2009 with development of the product line, version $n.m.0$ is released in 2010, and, one year later in an updated revision ($n.m.1$). In 2011, a new release is split up from $n.m.2$. Both releases now have to be maintained in parallel. Usually, the older release then primarily experiences corrective and maybe adaptive maintenance, whereas the newer one keeps evolving further (perfective, preventive maintenance). Until the end of maintenance of release m , considerable interaction between m and $m+1$ may occur (e.g., backporting of functionality), making complex configuration management (“*release management*”) tasks necessary between the product lines.

An even more profound break is the change of generations. According to [1] this is the case when the prospected changes are beyond a level of tolerance, so that it is necessary to reengineer the software architecture completely. This is

the case starting from year 2012 where generation $n+1$ is launched. The current state of the art in development and the lessons learnt from the previous generation build the input for creating a new product line architecture. Although the overall architecture is build from scratch, successful core assets of the prior generation may be ported (year 2014). Similarly as in a case of release change, maintenance operations performed on one product line generation may need to be transferred to the other one (e.g., forwardporting). This task is more challenging though, as, due to the differing product line architectures, the previously common core assets may have drifted apart considerably. This could be called “*product line generation management*”.

V. PRODUCT LINE EVOLUTION IN DEPTH

In the previous sections, we identified three types of “variability over time” and applied them to two evolution scenarios. Now, it is necessary to further decompose and to identify the relevant tasks and aspects to be considered within each type. This will serve as a foundation to define a research agenda for giving appropriate support with tools and methods in the future. For the type of product derivation, there already exists substantial research, even on advanced topics (e.g., staged configuration, software supply chains and ecosystems) [8], [11], [3]. For this paper, we will only perform this decomposition for “product line evolution”; addressing versioning-related variability issues is out of scope for this paper and will be future work.

Product line evolution just is starting to get a foundation suitable for modeling. In [26] a set of evolution operations is defined, and [5] presents initial concepts for evolution-enabled feature modeling. For defining the tasks and aspects relevant for product line evolution, we run through a complete iteration of what may be called the “*evolution lifecycle*”. Depending

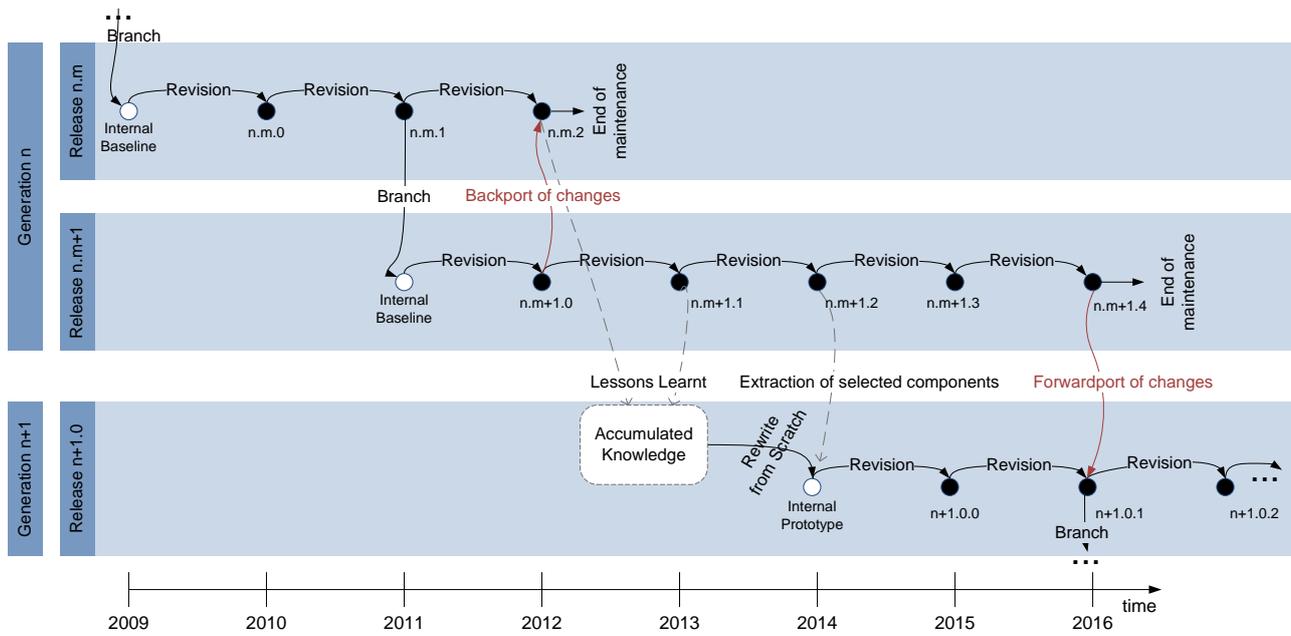


Fig. 2. Evolution paths of product lines (revisions, releases, and generations).

on the intention of an observer, product line evolution has the following lifecycle phases: future *planning*, present *tracking*, present/past *analysis*, and *correction/realignment*. Figure 3 illustrates this.

Proactive Planning. Scoping [25] is a crucial task for a product line; its future development has to be planned. This means, in the first case, planning the features and their variability for a certain moment in the future. This is however not the only relevant planning task. Additionally, it is necessary to plan the steps, how to reach the future plan (product line evolution). Next to the (proactive) evolution of the product line core assets, the maintenance of released products and previous product line releases and generations must be planned and aligned.

Tracking. Tracking a product line includes both logging its current state and the changes performed on it. When considering the state of a product line as its current features and their dependencies, this can be done by using variability modeling (e.g., feature modeling with versioned features and dependencies) and change recording. This tracks the maintenance/evolution of the product line variability. However, changes to released products (bug-fixes, updates) should consequently also be tracked. This might be difficult, for example in the context of software ecosystems and supply chains [3], [11], as a product line supplier might not even know about the actual products derived from its base product line for the end customers.

Analysis. Given that the product line evolution has been appropriately planned and tracked, we envision three types of analyses: evolution state analysis, evolution step analysis, evolution conformance analysis.

- Analyzing the *evolution state* means checking the consistency at a moment in time. This may involve that each feature on model level must have assigned implementation artifacts, or that dependencies on feature level may not contradict to those on implementation level.
- An *evolution step* is valid if it transforms one valid evolution state to another valid one. For one concrete evolution state as input, this is easy to check (simply applying the evolution step and checking the result for consistency). However, it might be possible to prove that an evolution step produces always a valid result given a valid input.
- Analyzing *evolution conformance*, finally, means checking whether the current and past evolution states and the transition steps performed comply to the evolution how it has been planned. The result of the evolution conformance analysis may not only comprise the actual deviation but also recommendations about evolution steps to perform to get on track again.

Each of these analyses will usually comprise the following tasks: gathering the data, performing the analysis, and reporting of the outcomes, for example by visualization. This also holds true for the maintenance of released products, which must be analyzed as well.

Correction and Realignment. Based on the results of the analyses it might be necessary to correct and realign the development efforts or the product line evolution plan.

As we address a complete iteration of the evolution lifecycle, we are confident that we cover the crucial tasks and aspects to be considered when defining a research agenda for product line evolution in the future.

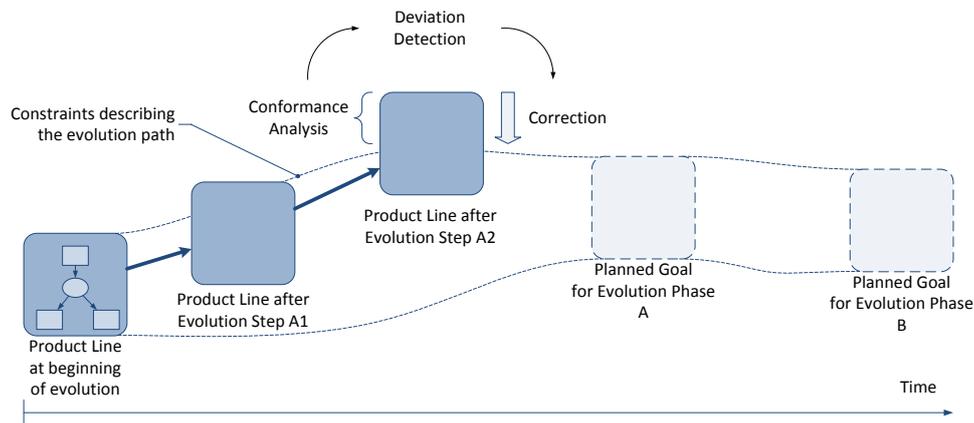


Fig. 3. Planning, tracking, analysis, and realignment of evolution.

VI. DISCUSSION

The aim of the paper is to perform first steps towards a common body of knowledge for variability over time. Validation of the results achieved is difficult, as a common terminology and understanding on concepts is missing. Addressing this problem best possible, we base our research both on existing work and exemplary validation. More precisely, our typification of “variability in time” is based on a literature survey and we referred to related work and assigned it to the corresponding types wherever appropriate. Second, we exemplarily validated the identified types by applying them to complex product line scenarios, where they exhibited expressiveness.

VII. CONCLUSION

Current authors address different time-related variability issues when talking about “variability in time”. Without a common framework of concepts, it will be difficult to relate advancements to each other, finally hindering progress. In this paper, we performed the first steps to approach this problem. We surveyed the different uses and found out that their notions are actually orthogonal. By generalization, we received three types of “variability in time”: variability of linear change over time (maintenance/evolution), multiple versions at a point in time (configuration management), and binding over time (product derivation). We applied the three types to two product line scenarios, where they exhibited expressive and discriminatory power, giving evidence for their usefulness as part of a body of knowledge for time-related variability issues.

As a first step towards further refinement, we extracted the necessary tasks and aspects relevant for the type maintenance/evolution, thereby covering a complete iteration of the “evolution lifecycle”. These results can be used for defining a research agenda for developing tools and methods supporting product line evolution. Obviously this paper can only be seen as a first step towards a common body of knowledge, which has to be developed in vital discussion with the research community. However, we hope that our work stimulates further researchers to engage with the various dynamic properties of product lines and the challenges imposed by this fact.

ACKNOWLEDGMENT

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre, <http://www.lero.ie/>.

We thank Christa Schwanninger for her valuable comments on a draft of this paper.

REFERENCES

- [1] M. Aoyama, “Continuous and discontinuous software evolution: aspects of software evolution across multiple product lines,” in *Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE '01)*. New York, NY, USA: ACM Press, 2001, pp. 87–90.
- [2] C. Atkinson, *Component-Based Product Line Engineering with UML*. Addison-Wesley, 2001.
- [3] J. Bosch, “From software product lines to software ecosystems,” in *Proceedings of the 13th Software Product Line Conference (SPLC '09)*, 2009, ISBN 978-0-9786956-2-0.
- [4] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, and K. Pohl, “Variability issues in software product lines,” in *Proceedings of the 4th International Workshop on Software Product-Family Engineering*. Heidelberg, Germany: Springer-Verlag, 2001.
- [5] G. Botterweck, A. Pleuss, A. Polzer, and S. Kowalewski, “Towards feature-driven planning of product-line evolution,” in *Proceedings of the 1st International Workshop on Feature-Oriented Software Development (FOSD'09)*. New York, NY, USA: ACM Press, 2009.
- [6] R. Capilla, A. Sánchez, and J. C. Dueñas, “An analysis of variability modeling and management tools for product line development,” in *Proceedings of the Workshop on Software and Services Variability Management. Concepts, Models and Tools*, 2007.
- [7] J. O. Coplien, “Multi-paradigm design,” Dissertation, Vrije Universiteit Brussel, 2000.
- [8] K. Czarnecki, S. Helsen, and U. W. Eisenecker, “Staged configuration through specialization and multilevel configuration of feature models,” *Software Process: Improvement and Practice*, vol. 10, no. 2, pp. 143–169, 2005.
- [9] S. Deelstra, M. Sinnema, J. Nijhuis, and J. Bosch, “COSVAM: A technique for assessing software variability in software product families,” in *Proceedings of the 20th IEEE International Conference on Software Maintainance (ICSM'04)*. Washington, DC, USA: IEEE Control Systems Magazine, 2004.
- [10] E. Dolstra, G. Florijn, M. de Jonge, and E. Visser, “Capturing timeline variability with transparent configuration environments,” in *Proceedings of the International Workshop on Software Variability Management*, May 2003.
- [11] H. Hartmann, T. Trew, and A. Matsinger, “Supplier independent feature modelling,” in *Proceedings of the 13th Software Product Line Conference (SPLC '09)*, 2009, ISBN 978-0-9786956-2-0.

- [12] A. Hubaux and A. Classen, "Taming time in software product lines," University of Namur, Rue Grandgagnage, 21, B-5000 Namur, Belgium, Tech. Rep., July 2008.
- [13] M. Jaring and J. Bosch, "Representing variability in software product lines: A case study," in *Proceedings of the 2nd Software Product Line Conference (SPLC '02)*. London, UK: Springer-Verlag, 2002, pp. 15–36.
- [14] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, Tech. Rep., Nov. 1990.
- [15] P. Knauber and J. Bosch, "Icse workshop on software variability management," in *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, vol. 0. Los Alamitos, CA, USA: IEEE Control Systems Magazine, 2003, p. 779.
- [16] P. Knauber and S. Thiel, "Session report on product issues in product family engineering," in *Proceedings of the 4th International Workshop on Software Product-Family Engineering*. Heidelberg, Germany: Springer-Verlag, 2001.
- [17] J. Koskinen, "Software maintenance costs. updated: Sept. 28, 2004." <http://users.jyu.fi/~koskinen/smcosts.htm>, P.O. Box 35, 40014-Jyväskylä, Finland, 2004.
- [18] B. P. Lientz and B. E. Swanson, *Software Maintenance Management*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1980.
- [19] D. Nestor, L. O'Malley, A. Quigley, E. Sikora, and S. Thiel, "Visualisation of variability in software product line engineering," in *Proceedings of the 1st International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, 2007. [Online]. Available: http://www.vamos-workshop.net/2007/files/VAMOS07_0038_Paper_7.pdf
- [20] L. Northrop and P. Clements, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
- [21] K. Pohl, G. Böckle, P. Clements, H. Obbink, and D. Rombach, Eds., *Product Family Development Seminar No. 01161*, April 2001.
- [22] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.
- [23] M.-O. Reiser and M. Weber, "Multi-level feature trees: A pragmatic approach to managing highly complex product families," *Requirements Engineering*, vol. 12, no. 2, pp. 57–75, 2007.
- [24] J. Savolainen and J. Kuusela, "Volatility analysis framework for product lines," in *Proceedings of the 2001 Symposium on Software Reusability (SSR '01)*. New York, NY, USA: ACM Press, 2001.
- [25] K. Schmid, "A comprehensive product line scoping approach and its validation," in *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. New York, NY, USA: ACM Press, 2002.
- [26] K. Schmid and H. Eichelberger, "A requirements-based taxonomy of software product line evolution," *Electronic Communication of the EASST*, vol. 8, 2007.
- [27] M. Sinnema and S. Deelstra, "Classifying variability modeling techniques," *Information & Software Technology*, vol. 49, no. 7, pp. 717–739, 2007. [Online]. Available: <http://dx.doi.org/10.1016/j.infsof.2006.08.001>
- [28] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch, "COVAMOF: A framework for modeling variability in software product families," in *Proceedings of the 11th Software Product Line Conference (SPLC '07)*. Heidelberg, Germany: Springer-Verlag, 2007.
- [29] M. Svahnberg and J. Bosch, "Evolution in software product lines: Two cases," *Journal of Software Maintenance*, vol. 11, no. 6, pp. 391–422, 1999.
- [30] M. Svahnberg, J. van Gorp, and J. Bosch, "A taxonomy of variability realization techniques," *Software - Practice and Experience*, vol. 35, no. 8, pp. 705–754, 2006.
- [31] C. Thao, E. V. Munson, and T. N. Nguyen, "Software configuration management for product derivation in software product families," in *Proceedings of the 15th Annual IEEE Int. Conf. and Workshop on the Engineering of Computer Based Systems*. Washington, DC, USA: IEEE Control Systems Magazine, 2008, pp. 265–274.
- [32] A. van der Hoek, "Design-time product line architectures for any-time variability," *Science of Computer Programming. Special Issue on Software Variability Management*, vol. 53, no. 3, pp. 285–304, 2004.
- [33] F. van der Linden, "Software product families in Europe: The Esaps & Café projects," *IEEE Software*, vol. 19, no. 4, pp. 41–49, 2002.
- [34] —, "Applying open source software principles in product lines," *UPGRADE – European Journal for the Informatics Professional*, vol. 10, no. 3, pp. 32–40, 2009.

Using Collaborations to Encapsulate Features? An Explorative Study

Martin Kuhlemann, Norbert Siegmund
Faculty of Computer Science
University of Magdeburg
Magdeburg, Germany
{mkuhlema,nsiegmun}@ovgu.de

Sven Apel
Department of Informatics and Mathematics
University of Passau
Passau, Germany
apel@uni-passau.de

Abstract—A feature is a program characteristic visible to an end-user. Current research strives to encapsulate the implementation of a feature in a module. Jak is a language extension to Java that allows programmers to encapsulate implementations of features in the form of a collaboration. In prior work, we and others faced problems when using collaborations in Jak and alike languages with too high expectations, e.g., to encapsulate widely scattered code of features such as transaction management in data bases. In this paper, we explore which criteria feature implementations must fulfill so that they can be encapsulated in Jak. The criteria that we found decisive are: *granularity* of code elements that should be encapsulated in a collaboration, *object-level extension* by features, and *object-oriented connections* of a feature’s code elements. We finally present a general guideline when to encapsulate a feature with a collaboration in Jak. Practitioners can now evaluate in advance whether Jak collaborations are suited to encapsulate their feature or not.

I. INTRODUCTION

Collaboration-based design (CBD) extends object-oriented design by the concept of a collaboration [1]. A *collaboration* is a module that encapsulates code fragments of an object-oriented program, e.g., statements, members, and classes. A collaboration can be composed flexibly with other collaborations to obtain complete software products. Usually, the code of one collaboration implements one complete feature, i.e., a user-visible program characteristic [2]. By flexibly composing collaborations, stakeholders can select features of a program [1].

Jak is a language that adds the collaboration concept to Java [1]. In prior work, we and others faced problems when we tried to encapsulate features with collaborations in Jak and alike languages, e.g., problems of granularity [3], parameter passing [4], or code replication [5]. Often the problems were encountered too late to make an easy design shift so the developers often refrained from reimplementing the features and instead tried to circumvent the problems (often with bad designs [3], [4]). We argue that developers can benefit from a guideline that describes for which kind of feature Jak collaborations are appropriate and for which features they are not. We propose such a guideline to avoid design flaws and reimplementation effort.

We conducted a case study and then we developed the guideline. In this study, we transformed implementations of a number of features from object-oriented Java code to

collaboration-based Jak code. The features we choose were implemented using object-oriented design patterns [6].¹ We choose them because their patterns occur in many places contributing to many feature implementations.

We developed general criteria which a feature must fulfill so that it can be encapsulated properly with a Jak collaboration. The guiding criteria that we found decisive are: *granularity* of code elements related to a feature, *object-level extension* by features, and *object-oriented connections* of features’ code elements. Based on these criteria we defined a guideline on how to use Jak-like collaborations. Developers can evaluate *in advance* whether collaborations can encapsulate the feature properly that is to be developed.

II. BACKGROUND:

COLLABORATION-BASED DESIGN WITH JAK

Jak adds support for collaborations to Java [1]. A collaboration encapsulates a set of classes and class refinements. A *class refinement* encapsulates members, which are added to classes or wrap methods of classes. Wrapping of methods allows programmers to add statements to the beginning and end of methods. The elements encapsulated in a collaboration can be composed with elements of other collaborations. Composing multiple collaborations finally synthesizes a compilable program.

In Figure 1, we show the two collaborations *Base* and *PointObserver*. *Base* contains a class *Point*. *PointObserver* contains a class refinement, which adds members and statements to the previously added class *Point*. It adds a field *observer* and a method *setObserver* to *Point* (Lines 7-10). Method *setY* of refinement *Point* (in *PointObserver*) refines method *setY* of class *Point* of *Base*, i.e., it adds statements to this method with an overriding mechanism (Jak’s overriding keyword is *Super*, Line 12). The refinement adds the subject role of the *Observer* design pattern to *Point*.

In our study we will analyze granularity of transformations. For that, we consider member and class introduction as well

¹Design patterns are descriptions of recurring development tasks and their according standard solutions [6]. A *pattern instance* is code that implements a pattern. Many Jak implementation approaches for patterns were introduced before [7]. However in [7], we did not evaluate Jak for implementing features but just compared mechanisms of Jak-like languages.

```

Collaboration Base
1 public class Point {
2   private int y;
3   public void setY(int newY) {
4     this.y=newY;
5   }}

Collaboration PointObserver
6 refines class Point {
7   private IObserver observer;
8   public void setObserver(IObserver newO) {
9     observer=newO;
10  }
11  public void setY(int newY) {
12    Super.setY(newY);
13    observer.update();
14  }}

```

Fig. 1. A sample class refinement in Jak.

as method wrapping *coarse-grained* transformations and transformations targeting statements or parts of statements as *fine-grained*.

III. CASE STUDIES

For different features, we encapsulated code, that implements a feature, into collaborations – each feature we selected for our study is implemented using an isolated pattern. For several features, we also applied collaboration concepts to reimplement the pattern’s solution (e.g., of wrapping). We argue that the structure of the studied features reoccurs in the structure of many other features because pattern implementations occur frequently and expose various shapes. If features are implemented without design patterns our criteria and guideline might not match.

Study setup: We encapsulated in collaborations code of features which are implemented by instances of the Gang-of-Four design patterns [6]. Specifically, in the three programs JHotDraw², Berkeley DB³, and Expression Product Line [8] we transformed object-oriented implementations of Gang-of-Four patterns into collaboration-based implementations.⁴ JHotDraw (30K lines of code) is a GUI framework; Berkeley DB (90K lines of code) is an embedded database engine for Java; Expression Product Line is an evaluator for mathematical expressions.

A. Results

We summarize interesting problems encountered during our study. We group these problems into categories *granularity*, *data type changes*, *object-level extension*, and *object-oriented connections between code elements*:

Granularity: In JHotDraw, an Adapter pattern instance is used to implement the *Undo* functionality for deleting figures. We analyzed different variants to implement this adapter with

²<http://sourceforge.net/projects/jhotdraw/>

³<http://www.oracle.com/database/berkeley-db/je/>

⁴When (a) transforming the whole implementation of a pattern instance includes repetitive tasks and (b) the effort to perform them all (manually) was too high then we concentrated on a sub-implementation requiring these tasks to be performed less often.

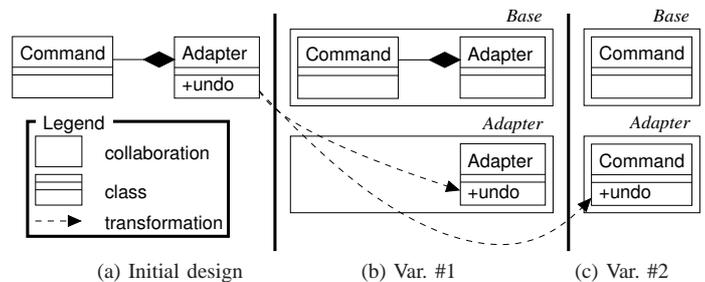


Fig. 2. Transforming the Adapter instance towards collaborations.

collaborations in which adapter methods are added with refinements (1) to an empty adapter class or (2) to the adapted class. With regard to variant #1 (Fig. 2b), we cannot separate calls to adapter methods into collaborations, which already encapsulate the adapter methods, because the calls were located in between other statements or were expressions inside other statements – Jak refinements however only may wrap methods. As a workaround, we added hook methods (or decomposed the methods in other cases) such that we can wrap these new methods. In variant #2 (Fig. 2c), we had to transform adapters that were composed from superclasses. We ended up with either few big collaborations, which replicate code, or with numerous small collaborations with less code replication but complex inter-dependencies.

Jak does not allow fine-grained extensions, e.g., adding formal parameters to methods, method calls to arbitrary methods, or changes to the return type. This caused problems for implementing instances of the patterns Bridge, Mediator, Observer, Prototype, and Template Method and hampered us or prevented us to implement Jak refinements.

Data type changes: To reuse code of an existing implementation of the pattern Composite, we had to change a field’s type. Jak does not support such transformation so we changed the code by hand (we changed a field’s type from Vector to List). If the transformed code would have been generic [9] our adjustment would have been less problematic.

Object-level extension: Strategy and State pattern instances allow developers to choose algorithms for objects of a *context class* [6]. In our study, we planned a design where we choose an algorithm by selecting a collaboration. The selected collaboration then should refine the context classes to add the strategy or state implementation statically to them. Unfortunately, strategy objects are polymorphic and assembled dynamically with delegation. So we cannot design an individual refinement which can be selected statically and which comprises one strategy or state algorithm. Additionally, references to objects of the pattern’s classes are set to null and tested for null and code is executed based on this test. These tests require object semantics and cannot be modeled with Jak-like collaborations. Hence, the transformations failed for Strategy and State.

For the pattern instance of Singleton, we faced problems when moving its code into a collaboration (replacing constructors with factory methods that are refined to return the same object at every call). We could not encapsulate all the

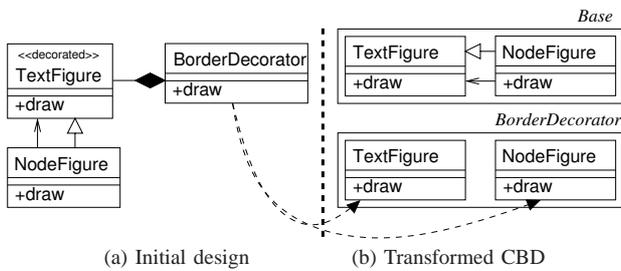


Fig. 3. Transformation of a Decorator instance (simplified).

```

1 refines class BinaryTreeLeaf implements VisitableNode {
2   public void accept(Visitor visitor) {
3     visitor.visitLeaf(this);
4   }}
(a)

1 refines class LineConnection {
2   public void visit(FigureVisitor visitor) {
3     visitor.visitFigure(this);
4   }}
(b)

```

Fig. 4. Name conflict prevents reuse for Visitor code.

feature-related code into the collaboration because we could not determine which code is related to the feature or depends on the pattern instance respectively. In particular, we could not determine execution paths which semantically rely on the pattern instance – a so-called feature mining problem [10].

Object-oriented connections between code elements: The object-oriented implementation of Decorator allows a decorator object to wrap objects of multiple classes which are connected by inheritance. When turning a decorator class into a refinement of a decorated class, we must replicate the refinement for every decorated class that can be instantiated. In the initial object-oriented design (Fig. 3a), objects of BorderDecorator class decorate objects of classes TextFigure and NodeFigure which both can be instantiated. However, refining both classes with the decorator’s code (Fig. 3b) applies the decoration twice accidentally for objects of NodeFigure. Objects of NodeFigure of which methods are decorated inherit and override methods of TextFigure which are decorated as well. A super call in the method NodeFigure.draw then causes the decoration to execute for TextFigure.draw and for NodeFigure.draw.

When we transformed the object-oriented implementation of pattern Proxy, more classes remained than we expected. That is, in the object-oriented design, single proxy objects wrap objects of different classes but some methods of the proxy objects do not wrap anything. Proxy methods that wrapped methods before became refinements of the methods they wrapped. Methods of the proxy class, which did not wrap anything, were not removed from the proxy class. But, as collaborations cannot encapsulate methods outside classes, the proxy classes remained to encapsulate these connected members. Collaboration names are no qualifiers of encapsulated code and cannot be used to reference this code.

For the Visitor pattern in our study we tried to reuse refine-

ments, which add accept methods and visitor classes, from an existing CBD Visitor implementation [7]. However, we failed reusing because methods and classes have incompatible names in JHotDraw and the existing implementation. Specifically, we could not reuse a refinement of class BinaryTreeLeaf (Fig. 4a) but had to implement a similar refinement for class LineConnection (Fig. 4b). We thus had to introduce a kind of code replication.

Other patterns: We were able to transform the implementations of Abstract Factory, Command, Facade, Factory Method, Flyweight, Interpreter, Iterator, and Chain of Responsibility to collaborations without new interesting problems. We faced problems for transforming the Memento instance but they do not allow us to conclude on the expressiveness of Jak [11].

IV. DISCUSSION

In our case studies, we faced more problems to implement or encapsulate the features with Jak than we expected:

Granularity: Most of the transformations were problematic or failed because Jak collaborations could not encapsulate fine-grained code elements, like formal parameters or method calls. Extensions to Jak that support fine-grained changes to code thus appear promising.

Object-level extension: Some patterns target to add a property to individual objects. A Jak-like refinement adds a property to all objects of a class but not just to individual objects. We argue that object-level properties conflict with class-level refinements, and thus CBDs as in Jak inherently cannot encapsulate instances of such patterns.

Object-oriented connections between code elements: We faced problems when different refined classes participate in the same inheritance hierarchy. We also observed a problematic balance between complex collaboration dependencies and code replication. Both are possible fields of future research.

Generally, we found it disturbing that a collaboration name is not a qualifier usable to reference the collaboration’s code. To use this code, we put it into publicly accessible classes (which can be referenced), although members in these classes are referenced from within one collaboration only. We argue that there are situations in which implementation details of a separated pattern instance should only be visible within the encapsulating collaboration or visible to features which explicitly extend this collaboration.

The transformations that we performed were simple but laborious.⁵ We had to implement these transformations manually because there is no sufficient support to restructure and transform code of CBDs.

A. Guideline

Now we give a *conservative* guideline so that developers can evaluate *in advance* whether Jak collaborations are suited to encapsulate a pattern instance. We allow cases in which Jak collaborations could be appropriate although our guideline

⁵For some object-oriented implementations the transformation into collaborations took up to three days (on average it took a few hours) because we had to restructure a major part of the program.

denies this issue, i.e., we allow false negatives. We found that Jak collaborations can encapsulate pattern instances whose implementation (1) is coarse-grained, e.g., whose implementation involves adding just classes and methods, (2) does not extend classes connected through inheritance, and (3) does not apply properties to individual objects of one class only. When pattern instances are known to become fine-grained, known to extend connected classes, or known to extend single objects of one class, Jak collaborations should not be used to encapsulate them.

V. RELATED WORK

Several researchers proposed languages similar to Jak, e.g., [12], [13]. Evaluating these languages is possible future work. We conjecture that the general criteria, which we found, are decisive to implement features in these languages, too.

Kästner et al. and Ye et al. observed that granularity is a decisive criterion when encapsulating features in modules [3], [14]. We confirm these results – we often had to add hook methods to encapsulate statements of method calls. In addition to prior work, we identified the criteria of *object-oriented connections* and *object-level extension* to be decisive for encapsulating features with Jak collaborations.

In another line of research, Kästner analyzed whether AspectJ is suitable for encapsulating features with aspects [15]. In contrast to his work, we concentrated on encapsulating code of various design pattern instances rather than code of self-chosen features. We argue to ensure this way that the features, which we encapsulated, cover the shape of a wide range of features. In addition, we determined general criteria that decide when Jak collaborations can encapsulate a feature.

Several researchers proposed to encapsulate design pattern instances in modules, e.g., [16], [17]. However, they all do not deal with CBDs and they all do not determine which *general* criteria decide language concepts' applicability. In prior work [7], we analyzed aspect-based implementations of patterns [17] and transformed them into Jak implementations. Here, we reuse the Jak implementation approaches (and tried to reuse code) side by side with new approaches in non-trivial programs to develop a general guideline for Jak.

VI. CONCLUSION

Collaborations are intended to encapsulate features in programs. In a study, we evaluated this aim for a variety of features implemented by design patterns. We found that collaborations can encapsulate some pattern implementations but not every implementation of collaborating program elements can be encapsulated with collaborations as in Jak.

By analyzing the problems we found, we identified criteria which guide whether Jak collaborations are well-suited to implement a particular feature: (1) the *granularity* of the code to encapsulate in a collaboration, (2) *object-level extension* by the feature to encapsulate, and (3) *object-oriented connections* between code elements of the feature to implement. Based on these criteria, we presented a guideline when to use collaborations in Jak-like languages in order to encapsulate a feature.

ACKNOWLEDGMENTS

The authors thank Don Batory and Maider Azanza for helpful comments on this work. Martin Kuhlemann was supported and partially funded by the *DAAD Doktorandenstipendium*, number D/07/45661. Norbert Siegmund was funded by the German Ministry of Education and Research (BMBF) in the ViERforES project (<http://vierfores.de/>), project number 01IM08003C. Sven Apel's work was supported in part by the German Research Foundation (DFG), project number AP 206/2-1.

REFERENCES

- [1] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 355–371, 2004.
- [2] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.
- [3] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proceedings of the International Conference on Software Engineering*, 2008, pp. 311–320.
- [4] M. Rosenmüller, M. Kuhlemann, N. Siegmund, and H. Schirmeier, "Avoiding variability of method signatures in software product lines: A case study," in *Workshop on Aspect-Oriented Product Line Engineering*, 2007, pp. 20–25.
- [5] S. Apel, "The role of features and aspects in software development," Ph.D. dissertation, Faculty of Computer Science, University of Magdeburg, 2007.
- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.
- [7] M. Kuhlemann, S. Apel, M. Rosenmüller, and R. E. Lopez-Herrejon, "A multiparadigm study of crosscutting modularity in design patterns," in *Proceedings of the International Conference Objects, Models, Components, Patterns*, 2008, pp. 121–140.
- [8] R. E. Lopez-Herrejon, D. Batory, and W. R. Cook, "Evaluating support for features in advanced modularization technologies," in *Proceedings of the European Conference on Object-Oriented Programming*, 2005, pp. 169–194.
- [9] S. Apel, M. Kuhlemann, and T. Leich, "Generic feature modules: Two-staged program customization," in *Proceedings of the International Conference on Software and Data Technologies*, 2006, pp. 127–132.
- [10] N. Loughran and A. Rashid, "Mining aspects," in *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2002, pp. 12–18.
- [11] M. Kuhlemann, "Transforming object-oriented design pattern structures into layers," Faculty of Computer Science, University of Magdeburg, Tech. Rep. 9, 2008.
- [12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *Proceedings of the European Conference on Object-Oriented Programming*, 2001, pp. 327–353.
- [13] S. Herrmann, "Object teams: Improving modularity for crosscutting collaborations," in *Proceedings of the International Conference NetObject-Days on Objects, Components, Architectures, Services, and Applications for a Networked World*, 2002, pp. 248–264.
- [14] P. Ye, X. Peng, Y. Xue, and S. Jarzabek, "A case study of variation mechanism in an industrial product line," in *Proceedings of the International Conference on Software Reuse*, 2009, pp. 126–136.
- [15] C. Kästner, "Aspect-oriented refactoring of Berkeley DB," Master's thesis, University of Magdeburg, Germany, Mar. 2007.
- [16] B. Meyer and K. Arnout, "Componentization: The Visitor example," *IEEE Computer*, vol. 39, no. 7, pp. 23–30, 2006.
- [17] J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002, pp. 161–173.

Modeling Variability of Augmented Software Product Lines

Johannes Müller
 Information Systems Institute
 University of Leipzig
 Leipzig, Germany
 jmueller@wifa.uni-leipzig.de

Abstract—Software product lines are an emerging paradigm enabling cost reduction while improving the overall quality of software products. This possibly allows an enhanced competitiveness and finally higher profits for a company, which establishes a software product line (SPL). One main subject of interest in SPL research is modeling and managing variability among systems of a SPL. Most research efforts address variability caused by technical differences between systems of a software family. However, recent marketing research reveals that customers perceive a product as a complex bundle of benefits. Beside technical features these benefits are also realized by non-technical features. With respect to the perceived value the paper relates the three sphere model of products to SPLs. It is argued that focusing on additional variability increases the profit of SPLs. Furthermore, additional sources of variability, such as license metrics, settlement options, revenue sources, services, and prices are discussed.

Keywords—Software Product Line; Feature Modeling; Marketing; Management; Economics

I. INTRODUCTION

The SPL research community has developed tools and techniques reducing the cost of software production while—at the same time—improving the quality of produced software systems. The major advantage of reduced costs is an improved competitiveness of a software business and thus an enhanced profitability ($\Pi = R - C$). However, the reduction of costs (C) is just one lever to increase profitability. Raising the revenue (R) the other. Revenue ($R = P * X$) depends on price (P) and quantity (X). Hence, setting the optimal price and enlarging the quantity of sold units increases the profit.

Marketing science works on understanding and capturing the value products provide to customers, who “see products as complex bundles of benefits that satisfy their needs” [1]. Marketing science distinguishes three spheres of a product to get a clear idea about what the customer really values on an offering. Fig. 1 depicts these three spheres *core benefit*, *actual product*, and *augmented product*.

The core benefit comprises the characteristics of a product, which address the underlying problem customers want to solve. For example a word processor solves the problem to perform writing activities faster, less error prone, and of higher quality. These core benefits have to be clearly communicated by the second sphere, the actual product. It is necessary that the offered features, the design, the name, and the packaging suggest that the product can solve

the problems of the customers. Finally, to create the most satisfying user experience, a company has to create an augmented product by adding complementary benefits to the actual product. In case of the word processor a 24/7 help desk is a complementary offer, which could support the writing efficiency of the customer. Other additional benefits might be flexible ways of payment.

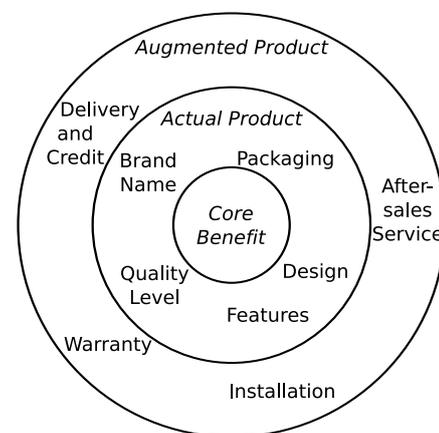


Figure 1. Spheres of a product [1]

The model of the three spheres is also useful for products of a SPL. In the first and second sphere the core benefits, the product’s features, and its quality level [2] are located. These are usually its functional and non-functional properties. How to model and to implement them within a family of software systems is subject of known research activities (cf. e.g. [3], [4]).

However, the augmented product, is not part of today’s SPL research even though “being able to choose between several variants can significantly increase the customers perceived value of a product” [5]. Therefore, adding complementary benefits to a SPL provides some chances to raise the profit of a SPL business. A SPL augmented with complementary offers is called an augmented SPL afterwards. The analysis of augmented SPLs comprises two questions:

- 1) Which additional offers enrich a SPL?
- 2) How to handle the variability introduced by these additional product characteristics?

The paper discusses both questions separately and wraps up with a discussion of related work and ongoing research.

II. EXTRA VARIABILITY INTRODUCED BY AUGMENTED SOFTWARE PRODUCTS

Literature about the management and marketing of digital products and software in particular (e.g. [1], [6]–[10]) mention especially five sources of variability, which are able to increase the customers perceived value of a software product. These different sources of variability are subsequently called *marketing features*. The variability caused by these features is called *augmented variability*. They are discussed afterwards.

License Metrics: License metrics are an integral part of software license agreements and define the units a price of a software product is applied to. Hence, they define what a user get per unit of price he or she pays [10]. According to [8] often used types of license metrics are *Concurrent User*, *Per Seat (named user)*, *Per Processor*, *Processor Core*, *Usage Based*, and *Financial Based*.

Settlement: The settlement can be done in two ways: recurring and non-recurring. Recurring settlement accounts software within a subscription model. The payment recurs in fixed time periods—i.e after certain number of transactions or similar units [7]. Non-recurring settlement accounts the final price to fully compensate the due.

Revenue Source: Revenue sources can be either direct or indirect [7]. Direct revenue sources comprise usage related sources such as payment per transaction or non-usage related sources such as connection fees or other license fees. Indirect revenue sources are mainly sponsored by advertising or by selling user related data.

Services: A service is “*any act or performance that one party can offer to another that is essentially intangible and does not result in the ownership of anything*” [11]. Software companies focus particularly on services such as support, maintenance, warranties, or software as a service. Offering additional customized services gives a company the opportunity to differentiate itself from its competitors in order to increase profitability [6].

Product Specification: The product specification process can be supported in two ways or by a combination of these ways: Either consultants assist the customers or the customers specify the product by means of a domain specific language (DSL). If consultants assist the customers, additional customers require additional personnel and generate costs—the approach does not scale. DSLs do not produce additional costs if more customers demand products of the product line—the approach scales. The price can help to convince the customers to use a DSL.

Price: The price (P) of a product influences the customer experience of a product and in most markets it is the ultimate mean of competition. However, the price is not an independent variable directly affected by the customer. Rather it is a dependent variable of all other features (f_i) of an augmented SPL ($P = f(f_0, f_1, \dots, f_n)$).

A promising approach to design $f(\dots)$ is value-based pricing [10], [12]. It relates the price to the customer’s perceived value of a product instead of to the production costs.

Nevertheless, the price needs to be modeled somewhere to track and document its relationship to the features of a product line. The following part discusses how to use existing modeling approaches for this task.

III. MODELING OF MARKETING FEATURES

A profound body of research has been done in modeling functional features of SPLs (e.g. [5], [13]–[15]). Therefore, it seems reasonable to start with these results to model marketing features. Nevertheless, an appropriate approach has to fulfill some requirements:

- Capture marketing features
- Link marketing feature and functional features
- Facilities to logically separate marketing- and functional features
- Constrain feature selection for unsupported combinations

In order to discuss market related issues of software product lines, it is important to distinguish between two related but different concepts:

- 1) Product line: “*is a group of products sharing a common, managed set of features that satisfy the specific needs of a selected market*” [16].
- 2) Product family: “*is a group of products that can be built from a common set of assets*” [16].

To foster reuse, a product line has to be based on one or more product families.

Marketing features of augmented software products are customer visible characteristics and therefore belong to a product line instead of to a product family behind.

A prominent variability modeling approach is feature modeling introduced as part of the Feature Oriented Domain Analysis (FODA) method [13]. Feature models capture and visualize variability of an SPL in terms of features. Other approaches are Orthogonal Variability Models [5], or decision tables.

One prominent part of feature models are feature diagrams. They capture commonalities and variability among systems of a product line in terms of features. A feature is “*a distinguishable characteristic of a concept [...] that is relevant to some stake holder of a concept*” [17]. Feature diagrams arrange features hierarchically. Features can be grouped. Features as well as feature groups can have cardinalities to restrict their occurrence in a final configuration [15]. Further diagram elements are feature attributes and feature diagram references [15]. These enumerated characteristics of feature diagrams are appropriate to model augmented SPLs.

A domain model of a SPL is augmented with marketing features by relating a supporting domain of marketing issues. The domain *marketing issues* comprises the variability and commonality depicted in Fig. 2.

Two cases can be distinguished. If the final product is marketable as a whole, the concept node gets a direct subfeature *marketing issues*. It covers all commonalities and variability introduced by augmenting the SPL. If the final product comprises features that are accounting units on their own, all marketable subfeatures get an

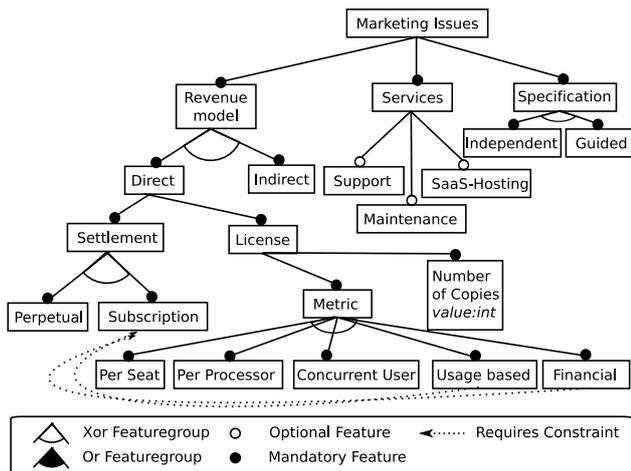


Figure 2. The concept marketing issues

additional feature *marketing issues*. Each specializes the feature *marketing issues* of the concept node, i.e. possible overlapping selections are overridden in the marketable subfeature.

For example, a customer orders a word processor but uses the mail merge functionality seldom. In this case, the customer could purchase the product on a per user basis with a perpetual fee but the mail merge functionality is accounted on its own and settled on a subscription model based on a usage based metric.

The legal consequences of all selected features of the feature *marketing issues* have to be combined in a final contract. This non-code artifact should be generated automatically to preserve the scalability of a SPL [18].

The additionally introduced marketing features are constrained by selected functional features and vice versa. To model restrictions, feature modeling offers *constraints* and *default dependency rules* [17].

Effective pricing often relies on market segmentation. One challenge is to identify the market segment of a specific customer. The pricing approach *versioning* is a method to let customers identify themselves by offering different versions of a product. All versions have some characteristics that are uniquely desired by customers of one market segment. The selection of some specific features of a product line should therefore reveal the market segment of a customer. These *marker features* can be used to classify the customers.

Modeling also marketing features of a SPL bloats the feature model and reduces its readability. Two existing concepts can help to reduce these effects by clearly separating the different addressed concerns. Feature model references [15] relate outsourced sub-feature-models by referencing their concept node in the source tree. The Feature Oriented Reuse Method (FORM) introduces layered feature models to separate different concerns of a product line architecture in distinct but related feature models [19], which are layered on each other. If combined, both can help to separate the technical from the marketing concerns

and therewith improve the readability and understandability of the overall model.

IV. RELATED WORK

The work [20] uses value-based techniques to improve the product specification process but does not consider value-based pricing. Helfrich et al. put marketing of SPLs onto the research agenda [21]. They also identify the importance of distinguishing the concepts of product families and product lines, when analyzing marketing related questions [22]. Kang et al. emphasize the importance of marketing for SPLs but mainly to improve the scoping of SPLs [23].

The work [15] indicates that feature models can serve as a source for pricing decisions. A pricing procedure for SPLs is introduced in [24] but it lacks of some characteristics for an effective use. Primarily, it neither exhibits facilities to concentrate on customer value to segment markets, nor takes it the discussed marketing features into account.

To align production planning by means of variation modeling to meet the business goals of a company is suggested in [25]. However, the authors ignore the variability introduced by augmenting software products.

V. CONCLUSION AND FUTURE WORK

The paper introduces augmented SPLs and suggests that considering marketing features can expand the turnover of a SPL. Further, different product characteristics are identified, which can augment the actual software product. Modeling these additional features with feature models was discussed.

The modeling of augmented SPLs is a further step towards an overall pricing method [26]. Feature models are promising to model augmented SPLs. Additionally, the contained data can serve as a basis for pricing a product of a SPL.

Further research is required to eventually develop a method to manage marketing features and to calculate prices almost automatically based on information provided by feature models. For that reason, tool support is of great importance. The appropriateness of existing feature modeling tools for pricing has to be surveyed. Another question is how to incorporate the modeling of marketing issues, especially pricing, into a overall domain engineering process.

The pricing of products is a delicate and even risky decision for every business. To foster the application of a pricing method for SPLs, it is vital that case studies or experiments are available in order to convince companies that the pricing method can improve their overall profitability.

REFERENCES

- [1] P. Kotler and G. Armstrong, *Principles of Marketing*, 11th ed. Upper Saddle River, NJ: Pearson/Prentice Hall, 2006.

- [2] L. Etzeberria, G. Sagardui, and L. Belategi, "Modelling variation in quality attributes," in *VaMoS '07*, K. Pohl, P. Heymans, K.-C. Kang, and A. Metzger, Eds. Limerick: Lero, Januar 2007.
- [3] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, 6th ed., ser. SEI Series in Software Engineering. Boston: Addison-Wesley Professional, August 2007.
- [4] F. J. v. d. Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.
- [5] K. Pohl, G. Böckle, and F. J. van der Linden, Eds., *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin: Springer, September 2005.
- [6] M. Kratochvil and C. Carson, *Growing Modular: Mass Customization of Complex Products, Services and Software*. Berlin: Springer, 2005.
- [7] A. Zerdick, A. Picot, K. Schrape, A. Artopé, K. Goldhammer, U. T. Lange, E. Vierkant, E. Lopez-Escobar, and R. Silverstone, *E-Conomics – Strategies for the Digital Marketplace*. Berlin: Springer, 2000.
- [8] o.a.V., "Key trends in software pricing and licensing: A survey of software industry executives and their enterprise customers," SIIA, Macrovision, SoftSummit, SCPMA, CELUG, Santa Clara, CA, Studie 2006–07, 2006. [Online]. Available: http://softsummit.com/pdfs_registered/SW_Pricing_Licensing_Report_20062007.pdf
- [9] G. Gruman, A. S. Morrison, and T. A. Retter, "Software pricing trends: How vendors can capitalize on the shift to new revenue models," PWC, Survey, jan. 2007.
- [10] T. T. Nagle and J. E. Hogan, *The Strategy and Tactics of Pricing: A Guide to Growing More Profitably*, 4th ed. Upper Saddle River, NJ: Prentice Hall, 2006.
- [11] P. Kotler, *Marketing management*, 11th ed. Upper Saddle River, NJ: Pearson/Prentice Hall, 2003.
- [12] S. R. Faulk, R. R. Harmon, and D. M. Raffo, "Value-based software engineering (vbse): a value-driven approach to product-line engineering," in *SPLC '00*. Norwell, MA, USA: Kluwer Acad. Pub., 2000, pp. 205–223.
- [13] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep., November 1990.
- [14] K. C. Kang, J. Lee, and P. Donohoe, "Feature-oriented product line engineering," *IEEE Software*, vol. 19, no. 4, pp. 58–65, 2002.
- [15] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.
- [16] J. Whitey, "Investment analysis of software assets for product lines," SEI, Carnegie-Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-96-TR-010, November 1996.
- [17] K. Czarnecki and U. W. Eisenecker, *Generative Programming Methods, Tools, and Applications*. Boston: Addison-Wesley, 2000.
- [18] J. Müller and U. Eisenecker, "The applicability of common generative techniques for textual non-code artifact generation," in *McGPLE, Nashville, TN, October 23, 2008*, no. MIP-0802. University of Passau, Oktober 2008, pp. 47–51.
- [19] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "Form: A feature-oriented reuse method with domain-specific reference architectures," *Annals of Software Engineering*, vol. 5, no. 1, pp. 143–168, 1998.
- [20] R. Rabiser, D. Dhungana, P. Grünbacher, and B. Burgstaller, "Value-based elicitation of product line variability: An experience report," in *VaMoS '08*, ser. ICB Research Report, P. Heymans, K. C. Kang, A. Metzger, and K. Pohl, Eds., 2008, pp. 73–79.
- [21] A. Helferich, K. Schmid, and G. Herzwurm, "Product management for software product lines: an unsolved problem?" *Commun. ACM*, vol. 49, no. 12, pp. 66–67, 2006.
- [22] —, "Reconciling marketed and engineered software product lines," in *SPLC '06*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 23–28.
- [23] K. C. Kang, P. Donohoe, E. Koh, J. Lee, and K. Lee, "Using a marketing and product plan as a key driver for product line asset development," in *SPLC '02*. London, UK: Springer-Verlag, 2002, pp. 366–382.
- [24] D. Sewerjuk, "Pricing of software product lines," in *MKWI '08*, M. Bichler *et al.*, Eds. Berlin: GITO, 2008.
- [25] G. J. Chastek and J. D. McGregor, "Modeling variation in production planning artifacts," in *VaMoS '09*, D. Benavides, A. Metzger, and U. Eisenecker, Eds., no. 29. Universität Duisburg Essen, Januar 2009, pp. 45–50.
- [26] J. Müller, "Towards a pricing method for software product lines," in *MKWI '10*. Univ.-Verlag Gött., 2010, to appear.

A Method Based on Association Rules to Construct Product Line Models

Alberto Lora-Michiels^{1,2} Camille Salinesi² and Raúl Mazo^{2,3}

¹ *Baxter International Inc, Lessines-Belgium*

² *CRI, Panthéon Sorbonne University, 90, rue de Tolbiac, 75013 Paris, France.*

³ *Ingeniería de Sistemas, Universidad de Antioquia, Medellín-Colombia.*

albertoloram@gmail.com, camille.salinesi@univ-paris1.fr, raulmazo@gmail.com

Abstract—*The success of a product line is the ability to improve application engineering, heavily depends on the quality of Product Line Models (PLMs). This paper reports on our effort to develop a method that exploits mining techniques such as the apriori algorithm, independence tests and the like to automate the construction of a PLM specified with FORE, starting from a collection of Product Models (PMs). Using these techniques, the proposed method guides the identification of candidate features, group cardinalities and dependencies. These can be used to progressively construct the PLM consistently with the existing PMs. The method was developed and tested in an industry setting starting with bills of materials as a collection of PMs. One interesting lesson learn from this experiment is that while the PLM is constructed, the domain engineer discovers errors in PMs. We believe that this advocates for a tighter intertwining between domain engineering and application engineering.*

I. INTRODUCTION

Product Line (PL) based development is a promising approach to develop software intensive systems in a reuse approach. Promises are multiple: reduced time to market, lower development costs, more trustworthy products, etc.

Approaches to construct PLMs are often focused on using clustering methods to elicitate, prioritize and triage requirements. Rather than a systematic process, the construction of an initial product line model from product requirement specifications somehow remains a “black art” and still mostly relies on the experience and expertise of domain engineers. Proposing new methods, techniques and tools that guide the construction of PLMs is thus a challenge [1].

Our method starts with a collection of PMs and produces PLMs in the FORE [2] notation. The method starts by arranging features of the collection of product models into a matrix of occurrences. Then, the process guides the construction of the general tree architecture by detecting candidate father-son dependencies, mandatory and optional relationships and completes it with group cardinalities. Last it guides the identification of other dependencies such as requires and excludes. The domain of statistics provides several mining techniques that could be used to support this process [3], [4], [5]. The research challenge was thus to identify which techniques could be used to efficiently detect the target items at each step of the method.

Our research strategy was to experiment the available techniques on a real case. Once a technique was detected,

further work was needed to identify with which parameter it should be used (e.g. thresholds). Last the overall method was evaluated on the complete case to identify how many models would be needed to obtain a “quasi-final” PLM.

The findings are: (i) cross table analysis used to determine exclude relationships; (ii) association rules analysis help identified mandatory and optional relationships; (iii) chi-square independence test combined with association rules are an effective way to identify require relationships; (iv) while constructing the PLM, errors are detected in PMs. The overall PLM construction process should thus be iterative and intertwined with PM correction; and (v) the techniques are efficient enough to be applied even on a large collection of PMs.

The rest of the paper is structured as follows. The next section presents mining techniques that we considered while developing the method. Section 3 presents our method and reports the rationale for the technique actually used at each step. Section 4 reports our evaluation in a real case. The concluding section presents related and future works.

II. MINING TECHNIQUES

Some mining techniques can be used in order to find relationships among a collection of variables. The better adapted to discover constraints between features are:

A. Cross Table Analysis

The cross table analysis consists in a paired based comparison among the different features. Normally, it is represented as a $n \times n$ matrix that provides the number of co-occurrences or conditional probabilities between features.

B. Association Rules

The objective of association rule mining [6] is the elicitation of interesting rules from which knowledge can be derived. Those rules describe novel, significant, unexpected, nontrivial and even actionable relationships between different features or attributes [7], [8]. Association rule mining is commonly stated as follows [9]: Let $I = \{i_1, i_2, \dots, i_n\}$ be a set of items, and D be a set of transactions. Each transaction consists of a subset of items in I . An association rule, is defined as an implication of the form $X \rightarrow Y$ where $X, Y \subseteq I$ and $X \cap Y = \emptyset$. Support, confidence, Chi square statistic and the minimum improvement constraint among others might be considered as measures to assess the quality of the extracted rules [10]. Support determines how often a rule is applicable to

a given data set of an attribute and it represents the probability that a transaction contains the rule. The confidence of a rule $X \rightarrow Y$ represents the probability that Y occurs if X have already occurred $P(Y/X)$; then it estimates how frequently items Y appear in transactions that contain X . Chi square statistics combined with its test (see next section) might be used as a measure to estimate the importance or strength of a rule from a given set of transactions and by this way to reduce the number of rules [11]. Finally the minimum improvement constraint measure not only indicates the strength of a rule but it prunes any rule that does not offer a significant predictive advantage over its proper sub-rules [12].

In this work, in the process for obtaining rules, we consider the Apriori Algorithm [9] that is supported on frequent item sets and is based on the following principle:

“If an itemset is frequent, then all of its subsets must also be frequent”
 Conversely “If an item set is infrequent, then all of its supersets must be infrequent to”.

For the purpose of this work items will be considered as features and transaction as PMs and the result of this pair wise is what we call the binary features matrix.

C. Chi Square and Independence Test

This test is based on Chi square value measure [11]. The measure is obtained by comparing the observed and expected frequencies, and using the following formula:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i} \quad (1),$$

where O_i stands for observed frequencies, E_i stands for expected frequencies, and i runs from $1, 2, \dots, n$, where n is the number of cells in the contingency table.

The value obtained in equation 1 is then compared with an appropriated critical value of Chi square. This critical value chi-square χ_0^2 depends of the degrees of freedom and level of significance. The critical value chi-square χ_0^2 will be calculated with $n - 1$ degrees of freedom and α significance level. In other words, when the marginal totals of a 2×2 contingency table is given, only one cell in the body of the table can be filled arbitrarily. This fact is expressed by saying that a 2×2 contingency table has only one degree of freedom. The level of significance α means that when we draw a conclusion, we may be $(1 - \alpha)$ % confident that we have drawn the correct conclusion (normally the α value is equal to 0.05). For 1 degree of freedom and a significance level of 0.05 critical value chi-square $\chi_0^2 = 3.84$.

The most common use of the test is to assess the probability of association or independence of facts. It consists on testing the following hypothesis:

Hypothesis null H_0 : The variables are independent.

Alternative hypothesis H_1 : The variables are NOT independent.

In every chi-square test the calculated χ^2 value will either be (i) less than or equal to the critical χ_0^2 value OR (ii) greater than the critical χ_0^2 value. If calculated $\chi^2 \leq \chi_0^2$ we conclude that there is sufficient evidence to say that cross categories are independent; otherwise can think on dependency.

III. PROPOSED METHOD

A. Method Overview

The main concerns in PLM construction are:

1) *Preparation*: to begin with our approach it is necessary to dispose of a collection of related features or artefacts for each application. Artefacts or features could be extracted from repositories and by means of clustering process the hierarchical relation could be established [3].

In another hand, a part generalization is required. Text mining techniques are used to deal with this generalization. In fact i.e Romanowski in [13] uses a neural network based text analysis program to generalize parts.

2) *Structural Dependency Identification*: to determine parent child relationships and also characterize which of them are mandatory and or optional;

3) *Transversal Dependency Specification*: to study the behavior among features that are not member of the parent child link and exploit not only all the possible mutual exclusive relationships known as “excludes”, but also distinguish all the relationships that indicate where a specific features may “require” the selection of another feature.

4) *Grouped Cardinality Specification*. Optional features that have the same father can be bundled, and constraints can be specified to indicate how many (at most and at least) features of the bundle can be selected together in a single product;

5) *Consolidation*. Results from previous concerns are evaluated by an expert.

Each of the following sections explain in which mining technique is proposed to support each of these phases

B. Preparation

Our approach is based on constructing a product line model based on existent product models. Then, to consider our approach and to successfully implement it, it is strictly required to get a collection of product models or related artifacts or features.

In order to execute our approach, we need a set of refinement relationships between features, that is, child-father tuples in two forms a list of relationships and its derivate matrix of feature occurrence in PM. This matrix is obtained by highlighting the features presence in product models.

- P1 , F1
- F1 , F5
- F1 , F6
- P2 , F1
- F1 , F6
- F1 , F4
- F1 , F5

	F1	F2	F3	F4	F5	F6	F7
P1	1	0	0	0	1	1	0
P2	1	0	0	1	1	1	0

b)

a)

Fig. 1. a) List of relationships. Column left represents fathers and right their childs. P1 is father of F1, F1 father of F5 and so on. b) PM -Feature binary matrix. The feature takes the value 1 if it is present in a product model and zero otherwise.. For instance F1, F4, F5, and F6 are present in P2 and contrary F2 and F7 are never taken into consideration by any product model.

C. Structural Analysis

From a collection of product models and their structure it is possible to determine i) bundles, parents and children; ii) the feature binary matrix, input for the association rule mining.

This step is handled by means of the Apriori algorithm to then obtain the mandatory and optional relationships.

Step 1: Identifying Structural Patterns. Due to the nature of input the identification of parents, sons and bundles consists on browsing this relational structure of a product model. The most representative example is the bill of material in manufactured finish good. The composition of the finish good is represented as a relational table that mainly integers the parent item and its components or children

Step 2: Running Association Rules Apriori Algorithm. Once the binary feature matrix is built, we have the input to apply the association rule data mining tool, that permit us not only to explore the relationships and dependencies but also to handle a huge amount of data in an optimal way. However, such algorithms developed are sometimes limited to the memory because of its size and calculus that they perform.

In fact the most complex task of the whole association rule mining process is the generation of frequent itemsets (in this part an itemset is considered as feature set). Many different combinations of features and rules have to be explored which can be a very computation-intensive task, especially in large databases. By setting the parameter association rule length equals to 1 for the Apriori algorithm, we can study only singles relations between features to avoid those computation complexities. Often, a compromise has to be made between discovering more complex rules and computation time.

To filter those rules that might be not valuable, it is important to calculate its support. As we have already seen, the support determines how frequent the rule is applicable to the product P. This value compared with the minimum support accepted by an expert (min support threshold), prunes the uninteresting rules.

To evaluate the interestingness and pertinence; that is it the reliability of the inference made by a rule, it is useful to evaluate its confidence. The task is now to generate all possible rules in the frequent feature set and then compare their confidence value with the minimum confidence (which is again defined by the expert). All rules that meet this requirement are regarded as interesting. All the final discovered associations with their support and confidence values, therefore, may be presented to stakeholders.

Furthermore the calculation of other measures is relevant to refine the process of selecting the appropriate association rule. For that we propose to calculate the Chi-square and to indicate the strength of a rule. The minimum improvement constraint measure not only gives us an idea about the strength but also prunes any rule that does not offer a significant predictive advantage over its proper sub-rules. This increases efficiency of the algorithm, but more importantly, it presents the user with a concise set of predictive rules.

Step 3: Identifying Mandatory Relationships Using Association Rules. Removing all association rules that do not satisfy the minimum improvement constraint, offers us the most relevant and significant rules available for the study.

It is obvious that those relationships that are always present in all the product models may be considered as mandatory. Now, if some ambiguous information is present in the database

and this one is not reliable at $\lambda\%$, in order to obtain mandatory relationships, the analyst may establish as a minimum confidence threshold the value $(100 - \lambda)\%$. Those rules whose confidence is greater than the $(100 - \lambda)\%$ may be considered as mandatory relationships. Bidirectional rules such as $F1 \rightarrow F2$ and $F2 \rightarrow F1$ may be also considered as mandatory relationships [14].

The relationship is classified as mandatory if at least one of the two properties mentioned before (high frequent features and bidirectional rules) occurs and, of course, the relationships belong to a parent child.

Step 4 Identifying Optional Relationships. Once parent child and as well mandatory relationships are identified the remaining parent child relationship may be classified as optional.

D. Transversal Analysis

By combining some results obtained from the previous sections such as the PM feature binary matrix and parental relationships with a cross tabulation analysis among features and an independence test to detect strong relationships, it is possible to identify exclude and requires relationships.

Step1: Identifying Exclude Relationships. Feature cross table display relationships between features. Let $F = \{F_1, F_2, \dots, F_n\}$ be a set of n features. $F \times F$ can be represented as a $n \times n$ cross table describing the joint occurrence between the feature i and j . When the joint distribution of (F_i, F_j) for all $i \neq j$ is equal to zero, that can be interpreted that there is no probability that F_i and F_j may occur at the same time. Thus, they are mutually exclusive and the relationship between F_i and F_j is considered as an exclude relationship.

A further analysis of contingency table could give us valuable information about some types of relationships such as mandatory, optional and requires.

Step2: Identifying Requires Relationships To identify requires relationships it is necessary to apply a Chi-square independence test. The test is performed for each single rule with 1 degree of freedom in order to prove with a significance level $\alpha = 0,05$ that the relationships between non parent-child features F_i, F_j for all $i \neq j$ are independent or not.

Thus, the association between F_i, F_j for all $i \neq j$ is considered as *dependent* if the χ^2 value for the rule with respect to the whole data exceeds the critical $\chi^2 = 3.84$ (χ^2 critical value with one degree of freedom and a significance level $\alpha = 0,05$) otherwise it is considered as *independent*.

E. Grouped Cardinality Analysis

This process helps the analyst in assigning the group cardinality value. It is interesting for the analyst to have a tool that allows him to estimate the cardinality for each non mandatory optional bundle.

Step1: Identifying All Possible Feature Sets for Each Bundle. All the possible optional features sets in each bundle are captured by browsing the product line model structure.

Step2: Counting feature's occurrences for each product model and optional bundle set obtained in step 1. Here we evaluate each PM and display how many features from the group are

present in the configuration. As a feature in our work is considered as a binary variable, by examining the presence of the group and the related features related in each product model, it is possible to obtain the group occurrence by adding their respective feature values.

IV. STUDY CASE

Our method was validated with the construction of the Baxter Bioscience Lessines product line model. Baxter Bioscience at Lessines-Belgium develops, manufactures and markets products for hemophilia and immune disorders.

To construct the packaging product line model, we focused our study around all the components that constitute the packaging process of the different treatments that Baxter Bioscience produces. We have worked with 536 packaging bill of materials (BOM) as product models and we have also handled more than 1500 items. After generalizing items, we proceed to apply our approach and evaluate the results obtained by estimating the algorithm time complexity and the scalability generating the desired constraints. First, examining the time complexity of the algorithm that supports our approach, we have observed that it is really efficient but it presents some limitations when studying group cardinalities. Second, performing a paired comparison of constraints generated from different random products samples (Fig. 2). We can observe structural dependencies show a high predictive capacity: 95% of the mandatory and optional relationships are founded when we take a random sample size of at least 350 products. The totality of the mandatory relationships are then discovered when the random sample size is greater than 450 products, however excludes and, especially, requires relationships, seem to depend to the problem size that is it, the number of constraints increases when sample size increases. This can be explained by examining the nature of the data used in our study case. Structural relationships mainly depend on the composition of the product; thus they depend of the parent child relationships or BOM composition and transversal dependencies are related to relationships attributes. More products means more attributes, and at the end, this means that more transversal relationships to be discovered.

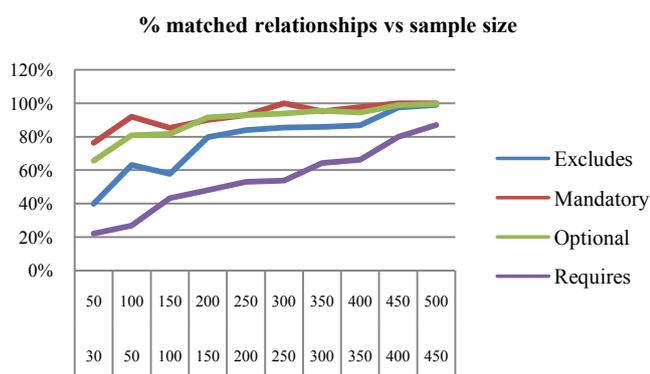


Fig. 2. Relationships matching (different sample size comparison)

V. CONCLUSIONS AND FUTURE WORKS

Our work is one of the first real scale experience of automation of the construction process of PLMs. To our

knowledge, it is one of the first approaches that integrates statistical techniques to identify commonalities and variabilities in a collection of a non predefined number of product models. Indeed, although rigorous, our proposal needs to be expanded and benchmarked with respect to alternative strategies explored, and implemented into a marketable tool.

Our experience showed that there is a need for a method that is able to deal with richer input information. For example, we had products that are defined with more complex than Boolean-type features, as for instance scalar variable (*e.g.* integer or real values as in performance characteristics of systems) or set variables (when system features can be instantiated a varying number of times in the same products). As a consequence, we believe that more complex relationships can be needed in the target PL models. How can these be specified? Remain still an open question for future researches. Several other fundamental questions are still open and their solutions are envisaged for future works. For instance: what is a good quality model to construct a product line model? How to deal with ambiguous information to construct a product line model? How to deal with more complex constraints? What statistical tools could be used to support the aforementioned questions?

REFERENCES

- [1] Marinelli F., de Weck O., Krob D., Liberti L., A General Framework for Combined Module- and Scale-based Product Platform Design, Second Internl Symp on Engineering Systems MIT, Cambridge, Mass, 2009.
- [2] Streitferdt, D.: FORE Family-Oriented Requirements Engineering, PhD Thesis, Technical University Ilmenau, 2004.
- [3] Chen K, Zhang W, Zhao H, Mei H. An Approach to Constructing Feature Models Based on Requirements Clustering. Internl Conf on Req Eng. pp 31-40 Paris 2005.
- [4] Moon S K., Kumara S R T., Simpson T W. Data mining and fuzzy clustering to support product family design, Proc of IDETC/CIE 2006.
- [5] Al-Otaiby T N, Alsharif M, Bond W. Towards Software Requirements Modularization using Hierarchical Clustering Techniques. 43rd Southeast regional Conference, Vol 2, Georgia, pp 223-228, 2005.
- [6] Ceglar, J.F. Roddick. Association mining. ACM Computing Surveys (CSUR), Vol 38 Issue 2, 2006.
- [7] Agar, B. and Kusiak, A, "Data-mining-based Methodology for the Design of Product Family," *International Journal of Production Research*, vol. 42, No. 15, pp. 2955-2969, 2004.
- [8] Jiao, J. and Zhang, Y., "Product Portfolio identification based on Association Rule Mining," *Comp.-Aided Design*, vol. 27, No. 149-172, 2005
- [9] Agrawal, R., Imielinski, T., Swami, A. "Mining association rules between sets of items in large databases." SIGMOD-1993, pp 207-216, 1993
- [10] Pang-Ning Tan, Michael Steinbach, Vipin Kumar, Introduction to data mining. Ch 6: Association Analysis: Basic Concepts and Algorithms. Addison Wesley 2006
- [11] Liu, W Hsu, Y Ma. Pruning and Summarizing the Discovered Associations. In ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD-99), August 15-18, San Diego, CA, USA, 1999
- [12] Bayardo, R. J.; Agrawal, R.; and Gunopulos, D. Constraint-Based Rule Mining in Large, Dense Databases. In Proc. of the 15th Int'l Conf. on Data Engineering, pp 188-197. 1999.
- [13] Romanowski, C.J. and Nagi, R., A data mining approach to forming generic bills of materials in support of variant design activities, ASME J of Computing and Information Science in Eng., 4(4), 316-328, 2004
- [14] Batory, D.; Thaker, S. Towards Safe Composition of Product-Lines. Dept. Computer Sciences, University of Texas, TR-06-33, 2006.

Measuring the Ability to Form a Product Line from Existing Products

Christian Berger, Holger Rendel, Bernhard Rumpe
 RWTH Aachen University
 Department of Software Engineering
 Aachen, Germany
 www.se-rwth.de

Abstract—A product line approach can save valuable resources by reusing artifacts. Especially for software artifacts, the reuse of existing components is highly desirable. In recent literature, the creation of software product lines is mainly proposed from a top-down point of view regarding features which are visible by customers. In practice, however, the design for a product line often arises from one or few existing products that descend from a very first product starting with copy-paste and evolving individually. In this contribution, we propose the theoretical basis to derive a set of metrics for evaluating similar software products in an objective manner. These metrics are used to evaluate the set of products' ability to form a product line.

Index Terms—software product line; software metrics; measurement; software architecture

I. INTRODUCTION AND MOTIVATION

Recent literature regarding the creation of software product lines often proposes to use end-user visible characteristics of several products which are referred to as features [1], [2]. In most cases, common and variable attributes of a set of products are identified and a feature model is created [3], [4]. This is a high-level view and supports a top-down method for implementing product lines which bases on the assumption that the code structure can and will be organized according to the identified features. In practice, however, it often happens that a product line is only set up after one or even several similar product variants are implemented. Hence, it is inevitable to not only look at the desired features but also at the existing implementation to identify potential for reuse. Therefore, a bottom-up method is necessary to look especially at the implementation of these artifacts to identify commonalities and differences which either support or prevent the setup of a product line from a set of similar products.

In the following we present an approach which uses the software architecture and existing software artifacts of a set of similar products to evaluate their potential to form a product line. This approach bases on a set of metrics for measuring the so-called *product line-ability* of the considered set of products.

II. RELATED WORK

The authors of [5] and [6] describe the importance of product line scoping which is a top-down view on a product line. Reusable assets of existing products can be identified by a product vs. feature-matrix which can be implemented using different methodologies like generative programming [3]. The

authors of [7] mention scoping as one aspect in number of steps when establishing a product line.

Metrics for evaluating product line architectures are discussed in several publications. The authors of [8] propose some metrics which are based on provided and required interfaces of components. However, these metrics are useful for object-oriented architectures only. A very formal specification of a product line architecture is given in [9] where parts of the architecture are treated as processes. In [10], some metrics are proposed to evaluate the quality of a product line which can only be applied for an existing product line with an already existing variability model.

The VEIA-project [11] also proposes very detailed metrics for product line architectures. Based on a function net and a feature model, these metrics measure the effort to integrate specific features into the product line. The use of function nets which define views on a so-called 150%-model of a product is also discussed in [12].

III. MEASURING THE PRODUCT LINE-ABILITY

In this section the theoretical basis for measuring the ability of a set of products to form a product line is outlined. Therefore, a set \mathcal{P} containing n similar products $p_1 \dots p_n$ is evaluated. Herein, the term *similar* needs to be precisely refined by a set of metrics which evaluate the considered products in an objective manner.

A. Specifying Similar Product Sets

As exemplarily shown in Fig. 1, a set \mathcal{P}_3 of three *similar* products p_1 , p_2 , and p_3 is shown for evaluating \mathcal{P}_3 's product line-ability. In this figure, three different classes named C_1 , C_2 , and C_3 of relations between two or more products are analyzed: C_1 describes the relation between two products, C_2 describes the reusability relation for commonly available parts for a specific product, and C_3 describes the reusability's benefit ratio for shareable parts for a specific product.

For evaluating a given set of similar products, each product is decomposed into $i = 1 \dots n$ so called reasonable *atomic* pieces $c_{p_j,i}$ for a concrete product p_j which is *self-contained* and *reusable*. We refer to these as *components* as defined in [2].

To perform a decomposition, all components $c_{p_j,i}$ must be identified and formally specified. Thus, we propose an

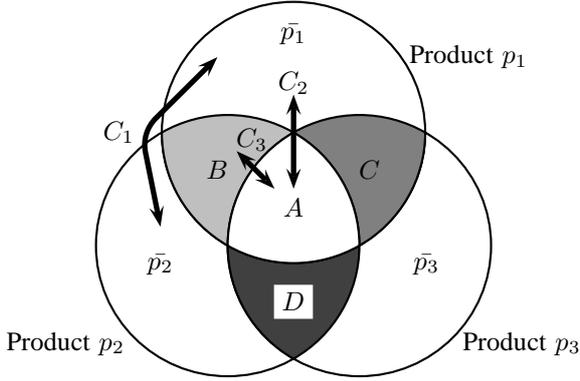


Fig. 1. Example for evaluating three similar products p_1 , p_2 , and p_3 . The circles indicate the set of components for each product; \bar{p}_2 denotes the complementary set of components for product p_2 without the sets B , A , and D . A denotes the set of components which are shared among all products; thus, all components in this intersection have at least a syntactically identical signature. B denotes all components which are shared only by p_1 and p_2 ; C and D are calculated in an analog manner. C_1 , C_2 , and C_3 denote different classes of relations.

annotated, directed graph G_{p_j} for product p_j which reflects the dependencies between all components $c_{p_j,i}$ which can for example be logical or communicative. The graph is defined as shown in Eq. (1).

$$\begin{aligned}
 G &:= (V, E) \\
 V &:= id \\
 E &:= V \times V \times \mathcal{P}(S) \times A \\
 S &:= id \times \{\mathbb{N}, \mathbb{R}, [\text{TYPE}], \dots\} \\
 A &:= \{0; 1\}
 \end{aligned} \tag{1}$$

As shown in Eq. (1), the directed annotated graph G consists of a set of ordered pairs of edges like $e_{1,2} = (c_1, c_2, (id, \mathbb{Z}), 0) \in E$. Each edge describes a formal dependency between the source component c_1 and its target component c_2 which reflects either a formal method call or a directed communication between component c_1 and c_2 . In the former case, it describes the required signature S in the target component for a successful method call, in the latter it defines a message which is sent from c_1 to c_2 containing the specified data in S . Components without any dependencies are so-called *isolated* components.

The set A can be used to define *required* and *optional* components within a product; a value of 1 defines an optional while 0 defines a required dependency. The former defines a component which is inherently necessary to fulfill a product's so called *basis functionality*, while the latter adds further functionality like convenience functions; if unspecified, the edge is regarded as *required*.

In Fig. 2, a graphical representation of the aforementioned definition for the graph is shown for a product of six components is shown. Here, $\bar{p}_{r_1} = K, L, M, Q$ describes one path of *required* components, while $\bar{p}_{o_1} = K, P, Q$ describes one path of *optional* ones. For calculating the set C_r of *required* and the

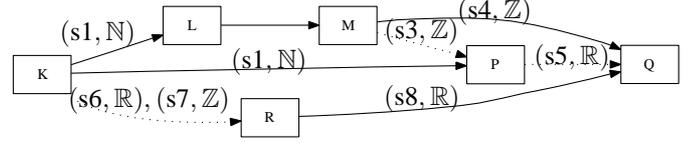


Fig. 2. An exemplary components' graph for six components K , L , M , P , Q , and R . The solid edges represent required communicative dependencies while the dotted edges represent optional ones. In this example, K sends the same message to L and P ; L sends an empty message to M and thus simply calls it. Moreover, K sends a message to R consisting of two data fields.

set C_o of *optional* components, recursive backtracking is used for all incident edges of an initially given set of components. Therefore, all product's components are initially added to the set C_o . Starting at a given required set of components C_{start} from the considered product which can be for example some components for an actuator, all edges to adjacent components are analyzed. If a required edge is found it is added to C_r which itself is analyzed recursively until all dependent required components are found. This set is finally subtracted from C_o . For example in Fig. 2 starting at Q , the following sets are calculated: $C_r = K, L, M, Q, R$ and $C_o = P$. The aforementioned algorithm does not identify isolated components because they do not contribute any reasonable data and thus, their relevance should be analyzed precisely.

B. Metrics for Evaluating the Product Line-Ability

For evaluating the product line-ability of a set of n similar products, the sets $C_{p_1,r} \dots C_{p_n,r}$ and $C_{p_1,o} \dots C_{p_n,o}$ with $\forall n : C_{p_n} \equiv C_{p_n,r} \cup C_{p_n,o}$ are calculated. Now, these sets can be evaluated according to Fig. 1. Therefore, different intersections between all sets are calculated which are used to evaluate different ratios and relations. For the sake of clarity, it is assumed that the denominator would not be 0 which means that two products do not share any components and thus, their comparison is not meaningful.

Size of Commonality.

$$SoC = \left| \bigcap_{i=1 \dots n} C_{p_i} \right| = \left| \bigcap_{i=1 \dots n} C_{p_i,r} \right| + \left| \bigcap_{i=1 \dots n} C_{p_i,o} \right|. \tag{2}$$

In Eq. (2), the *Size of Commonality* is shown which is calculated from set A in Fig. 1 containing the number of identical components. It can be calculated by comparing the components' signatures: Two components are syntactical identical if they have the same signature. If SoC is 0, no commonly reusable components could be identified. This comparison is called *syntactical signature identity* which is at least *necessary* but not *sufficient*. Therefore, *semantic signature identity* for two components must additionally be ensured which can be for example be evaluated automatically by using the component's test suites in an entangled manner which have to ensure path coverage at least.

Impact of Commonality.

$$IoC = \frac{|\bigcap_{i=1..n} C_{p_i,r}|}{SoC} \quad (3)$$

In Eq. (3), the *Impact of Commonality* is shown which relates *SoC* to all commonly shareable components. Obviously, the greater this ratio the more important are the commonly shareable components.

Product-related Reusability.

$$PrR_i = \frac{SoC}{|C_{p_i}|} \quad (4)$$

The ratio in Eq. (4) describes the reusability of *SoC* for a specific product p_i : The greater this ratio the better is its reusability. This ratio is denoted by C_2 in Fig. 1.

Impact of Product-related Reusability.

$$IPrR_i = \frac{|\bigcap_{j=1..n} C_{p_j,r}|}{|C_{p_i,r}|} \quad (5)$$

The ratio in Eq. (5) describes the impact of reusability of all commonly available components related to a specific product p_i which is also denoted by C_2 in Fig. 1. Here, the smaller $1 - IPrR_i$ for product p_i the greater is the impact of all commonly shared components for this product.

Reusability Benefit.

$$RB_{i,j} = \frac{SoC}{|C_{p_i} \cap C_{p_j}|} \quad (6)$$

In Eq. (6), the pairwise calculated *Reusability Benefit* is shown which is denoted by C_3 in Fig. 1. For example, this ratio for p_1 and p_2 is calculated by $\frac{|A|}{|A|+|B|}$. The greatest quotient among all products describes the pair which shares the least commonly available components and vice versa.

Relationship Ratio.

$$RR_{i,j} = \frac{|C_{p_i} \cap C_{p_j}|}{|C_{p_i} \cup C_{p_j}|} \quad (7)$$

In Eq. (7), the relationship between two products is calculated which is shown as C_1 in Fig. 1. Therefore, A together with the number of components which are shareable between these two products only is related to the joined set of all remaining components of both products; the greater $RR_{i,j}$ between two products p_i and p_j the more similar are both products.

Individualization Ratio.

$$IR_i = \frac{|C_{p_i} \setminus (\bigcup_{k=1..n, k \neq i} C_{p_k})|}{|C_{p_i}|} \quad (8)$$

In Eq. (8), the product-related *Individualization Ratio* is calculated which describes the product's individualization related

to the amount of components which are shared with at least one other product. The smaller this ratio the greater is this product's similarity with other products. In Fig. 1, this ratio is depicted by $IR_2 = \frac{|C_{p_2} \setminus (C_{p_1} \cup C_{p_3})|}{|C_{p_2}|}$ for product p_2 .

IV. APPLICABILITY OF THE METRICS

In the following, we apply the aforementioned metrics on a simplified example from the automotive domain for three different implementations of a door ECU. The first product as shown in Fig. 3 has only a lock/unlock functionality which locks the doors automatically at a specific vehicle's velocity. In Fig. 4, the product has no auto-lock function but power windows and a panic button to immediate closing in case of danger. Finally, in Fig. 5, a component exists to control window functions while opening or closing the hood of a convertible; this system also has an auto-lock function. All depicted signals have the same type.

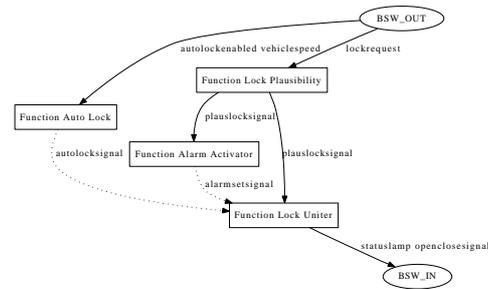


Fig. 3. Product p_1 "door ECU with auto-lock".

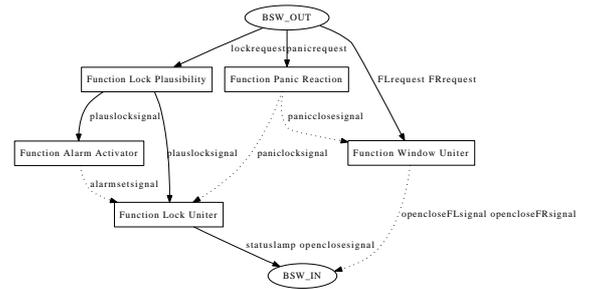


Fig. 4. Product p_2 "door ECU with power windows and a panic button".

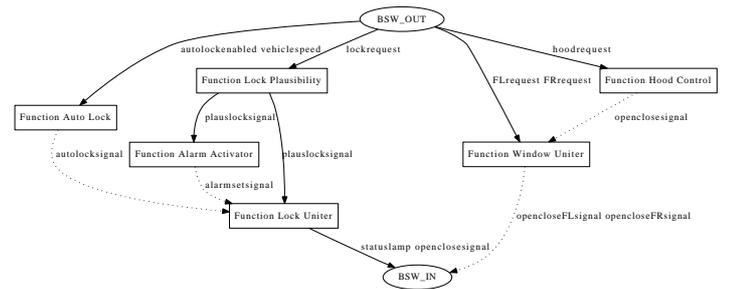


Fig. 5. Product p_3 "door ECU for convertibles".

To apply our metrics, we first have to determine the sets of products and their intersections. For the sake of clarity, the components are referred to by their abbreviation i.e. FLU for Function Lock Uniter. The *required* and *optional* components of the aforementioned products are shown in Tab. I.

TABLE I
REQUIRED AND OPTIONAL COMPONENTS

product	required	optional
p_1	FLP, FLU	FAL, FAA
p_2	FLP, FLU	FAA, FPR, FWU
p_3	FLP, FLU	FAL, FAA, FWU, FHC

Now we are able to map these components to the corresponding sets as depicted by Fig. 1 and shown in Eq. (9).

$$\begin{aligned}
 \bar{p}_1 &= \emptyset \\
 \bar{p}_2 &= \{FPR, FLU_{p_2}, FWU_{p_2}\} \\
 \bar{p}_3 &= \{FWU_{p_3}, FHC\} \\
 A &= \{FLP, FAA\} \\
 B &= \emptyset \\
 C &= \{FAL, FLU_{p_{1,3}}\} \\
 D &= \emptyset
 \end{aligned} \tag{9}$$

The application of different metrics yields the results summarized in Tab. II.

TABLE II
RESULTS OF METRICS FOR EXAMPLE PRODUCTS

	all	p_1	p_2	p_3	$p_{1,2}$	$p_{1,3}$	$p_{2,3}$
number of components		4	5	6			
SoC	2						
IoC	0.5						
PrR		0.5	0.4	0.33			
IPrR		0.33	0.33	0.33			
RB					1	0.5	1
RR					0.29	0.67	0.22
IR		0	0.6	0.33			

The results show that potential for reusability exists in general by *Size of Commonality*. *Impact of Commonality* has a value of 0.5 which means that the half of the common components are required. The product p_1 has to contribute to the product line because it has the highest *Product-related Reusability*. The *Impact of Product-related Reusability* is the same for all products and thus, no additional recommendation for a specific product to support the aforementioned ratio can be deduced. If *PrR* and *IPrR* for a specific product are small the product should not be part of the considered product line. The *Reusability Benefit* of p_1 and p_3 is the smallest because they share more than only the components of A . Besides, these products have also the highest *Relationship Ratio* which means they share the most common components if pairwise compared and thus, they are suitable for a product line. The ratio *IR* indicates that p_2 has the highest amount of

components which are independent from others. Hence, the product line should be created starting with the products p_1 and p_3 ; the product p_2 should be analyzed to identify potential for refactoring to improve its specific ratio of reusability.

V. CONCLUSION

This paper outlined a collection of metrics for measuring the ability for a product line of a given set of products. First, the mathematical basis was discussed to summarize the necessary information without relying on a particular model which can be code excerpts, UML sequence charts, or AUTOSAR functional components for example. Using the mathematical model, several metrics are presented and their importance and benefit for a product line are considered. In a simplified example, these metrics are exemplarily used to show their application.

Currently, these metrics are applied at an industrial project from the automotive domain that should be transformed into a product line. Here, the goals are to evaluate the proposed metrics, identify necessary and sufficient commonalities as well as correlations, and to estimate a set of values which recommends the creating of a product line. Another goal is to have a closer look on the models which describe software artifacts and their transformation into a suitable representation which we use as basis for the metrics.

REFERENCES

- [1] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
- [2] K. Pohl, G. Böckle, and F. Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.
- [3] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [4] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Technical Report CMU/SEI-90-TR-21, Software Engineering Institute - Carnegie Mellon University, Tech. Rep., 1990.
- [5] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.
- [6] I. John and M. Eisenbarth, "A decade of scoping - a survey," in *Proceedings of the 13th International Software Product Line Conference*, 2009.
- [7] P. C. Clements, L. G. Jones, J. D. McGregor, and L. M. Northrop, "Getting there from here: a roadmap for software product line adoption," *Commun. ACM*, vol. 49, no. 12, pp. 33–36, 2006.
- [8] E. Dincel, N. Medvidovic, and A. v. d. Hoek, "Measuring product line architectures," in *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*. London, UK: Springer-Verlag, 2002, pp. 346–352.
- [9] A. Gruler, M. Leucker, and K. Scheidemann, "Calculating and modeling common parts of software product lines," *Software Product Line Conference, International*, vol. 0, pp. 203–212, 2008.
- [10] T. Zhang, L. Deng, J. Wu, Q. Zhou, and C. Ma, "Some metrics for accessing quality of product line architecture," *Computer Science and Software Engineering, International Conference on*, vol. 2, pp. 500–503, 2008.
- [11] S. Mann and G. Rock, "Dealing with variability in architecture descriptions to support automotive product lines: Specification and analysis methods," in *Proceedings embedded world Conference 2009*. Nürnberg, Deutschland: WEKA Fachmedien, Mar. 3-5, 2009.
- [12] H. Gröniger, J. Hartmann, H. Krahn, S. Kriebel, L. Rothhardt, and B. Rumpel, "Modelling automotive function nets with views for features, variants, and modes," in *Proceedings of ERTS '08*, 2008.

A Custom Approach for Variability Management in Automotive Applications

Fabian Kliemannel, Georg Rock

PROSTEP IMP GmbH

Dolivostr. 11, 64293 Darmstadt, Germany

{fabian.kliemannel, georg.rock}@proststep.com

Stefan Mann

Fraunhofer Institute for Software and Systems Engineering ISST

Steinplatz 2, 10623 Berlin, Germany

stefan.mann@isst.fraunhofer.de

Abstract—Product lines are receiving more and more attention in software and mechanical engineering processes in the automotive industry. The benefits of using a product line approach are clearly defined in literature and widely accepted. Nevertheless, until now there are only few applications that exploit the variability knowledge during a complete development process. In this paper we argue that all the necessary prerequisites for the handling of variability are present. What is missing is on the one hand an integration of these techniques within the typical area specific development tools (as for example MATLAB/Simulink) and on the other hand the seamless transition of variant information between different engineering tools and development levels.

Index Terms—product line engineering; variability management; architecture descriptions; MATLAB/Simulink; automotive

I. INTRODUCTION

Product lines are receiving more and more attention in software and mechanical engineering processes in the automotive industry. The benefits of using a product line approach are clearly defined in literature and widely accepted. Nevertheless, until now there are only few applications that exploit the variability knowledge during a complete development process; from requirements engineering to the specification and execution of tests. We argue that all the necessary prerequisites for the handling or management of variability are present, as industrial tools like *pure::variants* [2] or the *Feature Modeling Plugin* [20] demonstrate. What we think that is missing is on the one hand a *complete integration* of these techniques within typical domain-specific development tools, and on the other hand the seamless transfer of variant information between different engineering tools. We previously addressed this problem by defining a reference process that allows for such an integrated approach [9], [10]. While we focused in this work on the early phases of system development as there are requirements engineering and architectural design, we want to extend this approach with respect to the implementation phase now. In this paper, we describe an exemplified integration of variability management within the domain specific engineering tool *MATLAB/Simulink* [14]. The extension of *MATLAB/Simulink*

This work was partially funded by the Federal Ministry of Education and Research of Germany in the framework of the EBASO project (German acronym for “engineering and assessment of variant rich embedded software”) under grant: 01IS09022. The responsibility for this article lies with the authors. For further information cf. the project’s website: <http://www.prostep.com/ebasol/>

was executed as close as possible to the usual development steps within Simulink. Thus, the user is not forced to be trained on a completely new tool. The additional functionality concerning the handling of variability as there are consistency checks, interactive configuration, dead feature detection, and switching between different variants of the Simulink model can be done without using an external variability management tool.

The rest of the paper focuses on the realization of this work, and thus constitutes a proof of concept for the before demanded complete integration of variability management within domain specific development tools.

II. REALISATION OF MODEL-BASED VARIABILITY MANAGEMENT

MATLAB/Simulink is a tool for modeling, simulating and analyzing multi-domain dynamic systems. The main part of Simulink is a graphical editing language based on block libraries which are used to specify dynamic systems. There are standard library blocks which can be used to model variability: *enabled subsystems*, *configurable subsystems* and *model variants*, for example. This was already described by the authors of [3], [7]. But the main purpose of these mechanisms is to model the behavior of single systems. Thus, we extend and reinterpret them to explicitly represent variation points in Simulink. The resulting Simulink extension, which we called “*v.control.mbd*”, consists of

- an additional block library for modeling variation points in Simulink models,
- a consistent data management, and
- a graphical user interface (GUI) for the configuration, analysis and assessment of variant-rich Simulink models.

Our block library consists of three types of variation points (Figure 1): There is an *optional subsystem* block, i.e. a subsystem which can be present in a configuration, but it does not have to be present. It can be turned *on* and *off* during a configuration step. The second type, an *XOR variation point* block, represents a subsystem which encapsulates subsystem alternatives. If the XOR variation point itself is present in a configuration, then exactly one of its children will be present. The third possibility is a combination of both mechanisms, i.e. an XOR variation point itself can be additionally optional.

These introduced variability mechanisms can be used within a hierarchical structure as usual within Simulink. Thus, it is

possible to specify an XOR variation point that again consists of XOR variation points, for example.

In the implementation of *v.control.mbd*, optional subsystems are realized using *enabled subsystem* blocks of Simulink connected with a *control* block (e.g. normalCB in Figure 6). An optional subsystem can be activated and deactivated by changing the respective value of its control block. XOR variation points are realized by standard *subsystem* blocks. Each subsystem alternative is again realized by an *enabled subsystem* block connected with a *control* block. It is ensured by the *v.control.mbd* implementation that only one alternative is active at a specific point of time.

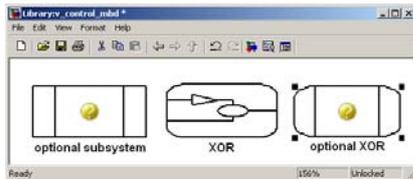


Figure 1. Elements of the *v.control.mbd* block library in Simulink.

The introduced variation points can be easily used within any Simulink model. The user simply has to insert a block from the *v.control.mbd* block library. The tool automatically creates the respective constraints and the needed variability data. This data is administered using a structure called *VarInfo* (see Figure 2). It contains the complete variability data of the current Simulink model and is stored in the *base workspace* of MATLAB. This approach allows to distinguish between model elements for the product line structure and model elements used to specify the functional behavior.

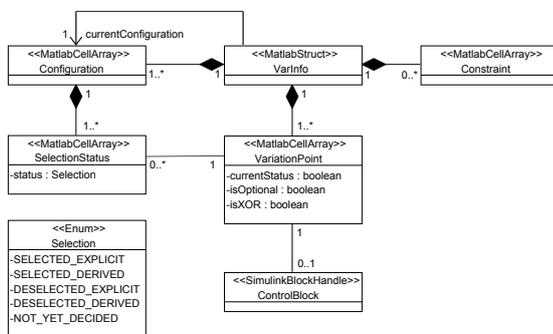


Figure 2. Data model of *v.control.mbd* in Simulink.

The third part of the *v.control.mbd* plugin is the configuration and analysis user interface shown in Figure 5. The configuration GUI is used to interactively specify a configuration of the Simulink model. A formal analysis engine is used during the configuration step in order to justify all the user made decisions and to check whether all the constraints are fulfilled.¹ Thus, the user is guided during the configuration process to guarantee that the resulting configuration is consistent and respects all the

¹A binary decision diagram (BDD) based engine is used in the current implementation.

automatically generated constraints [12]. Additional constraints can as well be added by the user in the GUI.

The following section presents a small example to show how the presented prototype is applied.

III. EXAMPLE APPLICATION OF THE V.CONTROL.MBD PLUGIN

To illustrate the before introduced concepts, a traffic lights product line is taken as an example. It consists of signalers for cars and for pedestrians. The traffic lights have additional right arrows to independently control cars turning to the right. Furthermore, there are members of our product line with signal buttons for pedestrians to request a green phase for them. Two alternative functionalities are specified for these signal buttons:

- an immediate switching to green, and
- a switching after a specific time period to ensure that cars will get a minimum time span for their green phase.

The corresponding variability is depicted in the feature model in Figure 3.

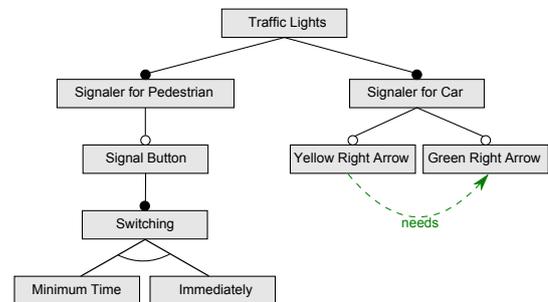


Figure 3. Feature model of a traffic lights product line.

Based on the feature model and its variability, a logical architecture (often called a *function net*) is specified in a next development step (Figure 4). It is a structural layout of the product line in the form of functions with input and output ports, communication relationships, and function variants [12]. The output of this modeling process is the starting point for the detailed specification of the behavior with the help of MATLAB / Simulink models.

Our prototypical tool *v.control.mbd* is capable of generating a variant-rich Simulink model out of function nets created for example in *v.control* [12], [13], [16], which is another prototype implemented in previous research activities. The result of this translation process is a Simulink model which provides the developer not only with a structural framework of the specified functions, but also with a mechanism to switch between different product line members within the Simulink model for simulation, validation or code generation purposes (see for example the translation of the XOR function *ahead_signaler* in Figure 4 into the corresponding Simulink specification depicted in Figure 6).

The graphical user interface (GUI) which is part of the *v.control.mbd* prototype (see Figure 5) allows the developer to interactively configure the Simulink model and to switch between different specified configurations. The first column

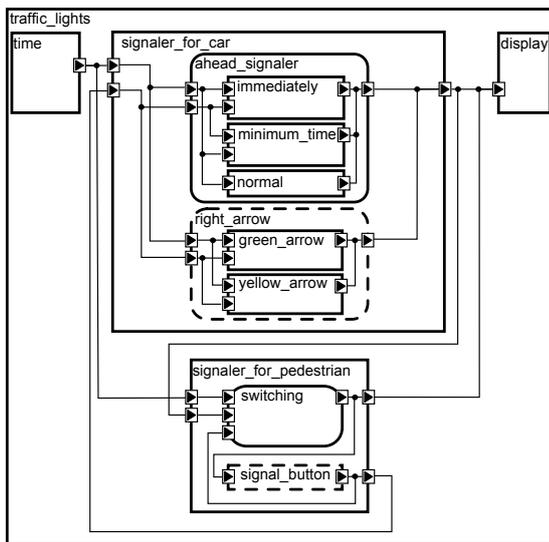


Figure 4. Function net of a traffic lights product line.

in Figure 5 shows a representation of the Simulink model structure. The further columns in the GUI represent different configurations. A row represents a subsystem (a function) in the model together with its corresponding configuration status. The configuration status can interactively be changed simply by clicking on the respective icon. The status of the corresponding subsystem in the Simulink model is automatically synchronized (see Figure 6). Thus, the configuration information is visible for the user within the Simulink model itself.

In order to simulate a particular configuration in Simulink, the user simply has to activate this configuration (e.g. the highlighted configuration *immediately* in Figure 5) and simulate the model.

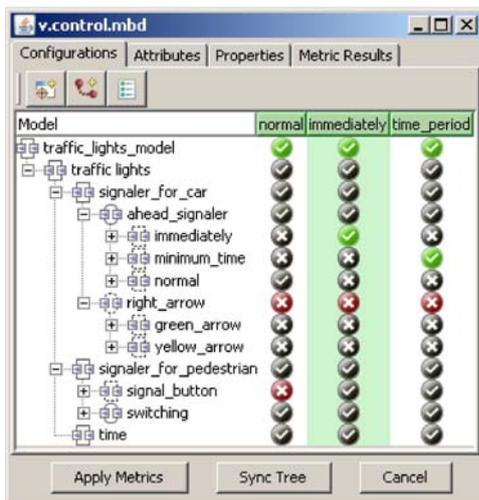


Figure 5. User interface for the configuration of a Simulink model.

If a configuration is changed by clicking on a status in the configuration GUI, *v.control.mbd* adjusts the selection status of all directly or indirectly affected systems. This way

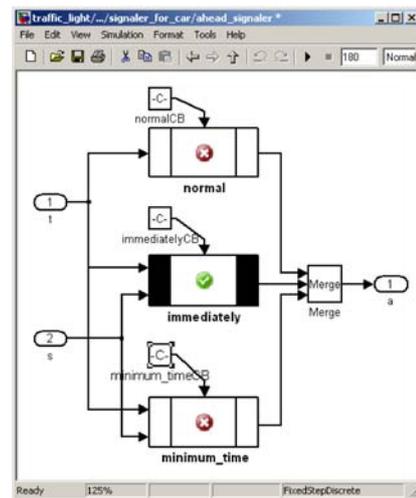


Figure 6. The configured Simulink model (extract): XOR variation point *ahead_signaler*.

it is ensured that variability constraints resulting from the architectural design or from the user input are continuously valid. All of these features (configuration, consistency check, constraints editing, simulation) can be done in MATLAB/Simulink using the *v.control.mbd* plugin.

IV. RELATED WORK

The main focus of our activities is the application of product line principles together with a step-wise, function-oriented, and architecture-centric development. Such a modular and compositional approach is necessary to tackle the complexity found in the automotive domain as described in [17], for instance. AUTOSAR [1] becomes a standard in this domain: Besides the standardization of basic software in vehicles, it also consists of architecture description languages (ADLs) for the specification of application software and hardware. But it does not prescribe the development methods nor how to support variability management and product line engineering. There are other ADLs and methods which can be used for development steps before AUTOSAR, e.g. [15], [8], [19]. The latter also aims to directly support the development according to the AUTOSAR philosophy.

There are already several surveys of the usage of MATLAB/Simulink for product line engineering. In [5] it was investigated how code generators interpret Simulink constructs which were used for modeling variable parts. The construction of a product model on the basis of model libraries and templates by selecting features was investigated in [11]. Which Simulink constructs are suitable to express variability, how they can be configured, and how product models can be derived from such models is investigated in [3], [7]. The approach we presented here is very similar to this work. However, we have chosen to realize a deep integration of the variability handling mechanisms within Simulink which allows for a stand-alone usage of the new functionality in Simulink as well as for the integration of Simulink within an architecture-centric

development methodology [9].

Model elements in application models are annotated with features in [6]. The features are used to specify *presence conditions* for the elements. The approach how application models (e.g. UML class diagrams, Simulink models etc.) are configured with the help of features is similar to our work. Also, we use an analog approach for the verification of feature models. But in contrast to [6], we have explicitly integrated the notion of variability into our application models [12]. Thus, we are able to configure and analyze the variability in application models without an external feature model.

In [18] a catalog of verification properties that are essential to a type safe composition of modules are introduced on the basis of a formal interpretation of feature models. However, they assume that each feature is implemented by a distinct module, which is not the case in our approach.

A similar approach for the configuration of variant-rich (architectural) models with the help of a feature model is implemented in [4]. As in [12], they integrate a feature model with the other models by building an internal, unified feature model which is used for an interactive configuration with feedbacks.

V. CONCLUSION

Development tasks in the automotive industry are usually very complex. Not only the complexity, but also the huge size of data to be handled during a complete product development lead to the necessity to introduce new development methods that tackle these problems. One of the most effective way to overcome the described complexity and size problem is the proactive, planned and optimized reuse of development artifacts. A high grade of reuse can be reached using the development paradigm of product lines. A necessary prerequisite to implement reuse within the development artifacts is the possibility to express and analyze variability already during the development. The approach presented in this paper allows for such an integrated and early specification and analysis of variability without adding a new paradigm or tool. Based on the general ideas of feature modeling it is possible to marry stand-alone applications extended to handle variability with the overall development process as shown in the paper. Although we present only one possible witness for such an integration, we think that it could be extended to many engineering tools in general. This would lead to a pervasive variability management approach that will find acceptance within the industrial development departments.

REFERENCES

- [1] AUTOSAR Development Partnership, "AUTOSAR – AUTomotive Open System ARchitecture." [Online]. Available: <http://www.autosar.org/>
- [2] D. Beuche, "Modeling and building software product lines with pure::variants," in *Proc. 12th Int. Software Product Line Conf. (SPLC 2008)*. Limerick, Ireland: IEEE Computer Society, Sep. 8–12, 2008, pp. 358–358.
- [3] D. Beuche and J. Weiland, "Managing flexibility: Modeling binding-times in Simulink," in *Proc. 5th European Conf. on Model Driven Architecture – Foundations and Applications (ECMDA-FA 2009)*, R. F. Paige, A. Hartman, and A. Rensink, Eds. Enschede, The Netherlands: Springer-Verlag, Jun. 23–26, 2009, pp. 289–300, LNCS 5562.
- [4] G. Botterweck, A. Polzer, and S. Kowalewski, "Interactive configuration of embedded systems product lines," in *Proc. Int. Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2009)*, San Francisco, California, USA, Aug. 24, 2009, colocated with the 13th Int. Software Product Line Conf. (SPLC 2009). [Online]. Available: <http://www.lero.ie/maple2009/>
- [5] S. Bunzel, U. Judaschke, and E. Kalix, "Variant mechanisms in model-based design and code generation," in *Proc. MathWorks Int. Automotive Conf. (IAC 2005)*, 2005.
- [6] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness OCL constraints," in *Proc. 5th Int. Conf. on Generative Programming and Component Engineering (GPCE 2006)*. Portland, Oregon, USA: ACM, Oct. 22–26, 2006, pp. 211–220.
- [7] C. Dziobek, J. Loew, W. Przystas, and J. Weiland, "Von Vielfalt und Variabilität – Handhabung von Funktionsvarianten in Simulink-Modellen," *Elektronik automotive*, pp. 33–37, Feb. 2008, (title in English: "Model diversity and variability – handling of functional variants in Simulink models").
- [8] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (AADL): An introduction," CMU-SEI, Technical Report CMU/SEI-2006-TN-011, Feb. 2006. [Online]. Available: www.aadl.info
- [9] M. Große-Rhode, "Architecture-centric variants management for embedded systems. Results of the project 'Distributed Development and Integration of Automotive Product Lines'," Fraunhofer ISST Berlin, ISST-Report 89/08, Oct. 2008. [Online]. Available: <http://veia.isst.fraunhofer.de/>
- [10] M. Große-Rhode, S. Euringer, E. Kleinod, and S. Mann, "Rough draft of VEIA reference process," Fraunhofer ISST Berlin, ISST-Report 80/07, Jan. 2007. [Online]. Available: <http://veia.isst.fraunhofer.de/>
- [11] S. Kubica, "Variantenmanagement modellbasierter Funktionssoftware mit Software-Produktlinien," Ph.D. dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, Institut für Informatik, 2007, Bd. 40, Nr. 4.
- [12] S. Mann and G. Rock, "Dealing with variability in architecture descriptions to support automotive product lines," in *Proc. 3rd Int. Workshop on Variability Modeling of Software-intensive Systems (VAMOS 2009)*, D. Benavides, A. Metzger, and U. Eisenacker, Eds., Sevilla, Spain, Jan. 27–30, 2009, pp. 111–120, ICB-Research Report No. 29, ISSN 1860-2770 (Print); 1866-5101 (Online).
- [13] —, "Dealing with variability in architecture descriptions to support automotive product lines: Specification and analysis methods," in *Proc. embedded world Conference*. Nürnberg, Germany: WEKA Fachmedien, Mar. 3–5, 2009, ISBN 978-3-7723-3798-7.
- [14] "Simulink 7," CASE tool, The MathWorks, 2009, a MATLAB toolbox, see <http://www.mathworks.com/>.
- [15] J. K. R. van Ommerring, F. van der Linden and J. Magee, "The Koala component model for consumer electronics software," *IEEE Computer*, pp. 78–85, Mar. 2000.
- [16] G. Rock and S. Mann, "Assessment of product line architecture descriptions in v.control," in *Software Quality Engineering – Proc. of the CONQUEST*, I. Schieferdecker and S. Goericke, Eds. Nürnberg, Germany: dpunkt.verlag, Sep. 16–18, 2009, pp. 163–178, ISBN 978-3-89864-637-6.
- [17] J. Schäufler and T. Zurawka, *Automotive Software Engineering*, ser. ATZ-MTZ-Fachbuch. Vieweg, 2003.
- [18] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe composition of product lines," in *Proc. 6th Int. Conf. on Generative Programming and Component Engineering (GPCE 2007)*. Salzburg, Austria: ACM, Oct. 1–3, 2007, pp. 95–104.
- [19] The ATESSST Consortium, "EAST ADL 2.0 specification," Specification 2008-02-29, 2008, draft. [Online]. Available: www.atesst.org
- [20] "Feature modeling plugin (version 0.7.0)," CASE-Tool (Eclipse-Plugin), University of Waterloo, Canada, 2006, see <http://gsd.uwaterloo.ca/projects/fmp-plugin/>.

Introducing TVL, a Text-based Feature Modelling Language

Quentin Boucher, Andreas Classen,* Paul Faber and Patrick Heymans

PRECISe Research Centre
Faculty of Computer Science
University of Namur
5000 Namur, Belgium

Email: {qbo,acs,pfaber,phe}@info.fundp.ac.be

Abstract—Feature models are a common way to represent variability in software product line engineering. For this purpose, most authors use a graphical notation based on FODA. The main drawback of those approaches is their lack of scalability: they generally do not fit real-size problems. Indeed, their graphical syntax does not account for attributes or complex constraints and becomes a burden for large feature models.

In this paper, we present TVL, a text-based feature modelling notation that is both light and comprehensive, meaning that it covers most constructs of existing languages, including cardinality-based decomposition and feature attributes. The main objective of TVL is to provide engineers with a human-readable language supporting large-scale models through modularisation mechanisms. Furthermore, TVL can serve as an extensible storage format for feature modelling tools. We illustrate the various concepts of the language with short code fragments.

I. INTRODUCTION

In software product line engineering (SPLE), Feature Models (FMs) are a common means to represent the variability of a software product line (SPL) [1]. Almost all existing FM languages are graphical notations based on FODA Feature Diagrams (FDs) which were introduced in the seminal paper by Kang *et al.* [2]. Since this original proposal, several extensions have been proposed by various authors. In all those dialects, FMs are represented as trees whose nodes denote features and whose edges represent top-down hierarchical decomposition of features. Consider the example FD in Figure 1 modelling a product line of personal computers. The *Computer* consists of a *Motherboard*, a *CPU*, a *Graphic Card* and some *Accessories*, which are optional (indicated by the hollow circle); all of these features are further decomposed. In addition, although not shown in the figure, each of the features has a *price*, which can be modelled as an attribute [3].

While such a graphical representation is supposedly more accessible to non-technical stakeholders, we believe that working with large industry-size FDs can become a tricky task for several reasons. First, to create a large FD, the graphical syntax is a burden that cannot be mastered without dedicated tool support (though many FM tools use directory tree-like representations themselves). Secondly, given that a FD is a tree on a two-dimensional surface, there will inevitably be large physical distances between features, which makes it hard to

navigate, search or interpret the FD. Finally, most notations do not have graphical means to represent constructs like attributes and constraints which are essential for industrial FMs.

Even though future tools might solve some of these issues, they would just attack the symptoms of the underlying problem. We attack its cause and propose to change the medium of the notation to just *plain text*. We do not question the need for a graphical representation, we rather propose an alternative and complementary textual notation. This would allow one to view and edit FMs either graphically or textually, depending on one's skills and preferences. We thereby hope to facilitate the dissemination of FMs in industrial settings. The goals for the new language are to be *scalable* by being *succinct* and *modular* as well as *comprehensive*.

Our language is called TVL, for *text-based variability language*. Plain text has a number of advantages the most important of which is the abundance of established tools dealing with text, generally program code. Moreover, the syntax of TVL is inspired by the syntax of C and should appear intuitive to any engineer who has come in contact with one of the many programming languages with a C-like syntax. We believe that these choices will also ease acceptance of FMs in industrial contexts because TVL is similar to the languages used in such environments and because it does not need dedicated modelling tools to be deployed.

TVL is *scalable* because it is *succinct* (its syntax is very light, as opposed to XML, for instance) and because it offers a number of mechanisms for *modularisation* and separation of concerns. The language is *comprehensive* because it integrates most of the FM constructs proposed in the twenty years since the advent of FODA.

At this stage, TVL is a language proposal. It is formally defined with an LALR grammar, a formal semantics [4] and comes with a reference implementation available online.¹ It is meant to be a basis for discussion and we are mainly interested in feedback about the language and its syntax.

The remainder of the paper is structured as follows: Section II introduces the TVL syntax, we survey related work in Section III and conclude in Section IV.

*FNRS Research Fellow

¹Download at the TVL website <http://www.info.fundp.ac.be/~acs/tvl>.

II. SYNTAX

In this section we present an overview of the TVL syntax using code snippets. The formal BNF grammar is available online. The different sub-sections introduce five major parts of the language i.e. features, attributes, expressions, constraints and modularisation mechanisms.

The different concepts of TVL will be illustrated using a basic personal computer product family example FD introduced in Section I and shown in Figure 1.

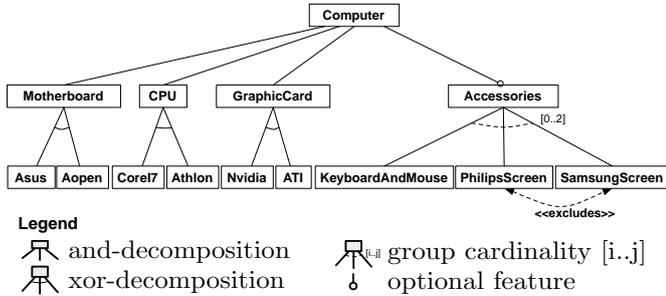


Fig. 1. Computer example FD

A. Feature hierarchy

The TVL language has a C-like syntax: it uses braces to delimit blocks, C-style comments and semicolons to delimit statements. The rationale for this syntax choice is that nearly all computing professionals have come across a C-like syntax and are thus familiar with this style. Furthermore, many text editors have built-in facilities to handle this type of syntax.

In our example, the root feature, *Computer*, is decomposed into four sub-features by an *and*-decomposition: *Motherboard*, *CPU*, *GraphicCard* and *Accessories*. Furthermore, the *Accessories* feature is optional while the other three features are mandatory. In TVL, this is written as follows:

```
root Computer {
  group allOf {
    Motherboard,
    CPU,
    GraphicCard,
    opt Accessories
  }
}
```

A decomposition type in TVL is defined with the **group** keyword. Predefined decomposition operators are **allOf**, as used in this example for an *and*-decomposition, **oneOf** for *xor*-decompositions and **someOf** for *or*-decompositions. It is also possible to specify a cardinality-based decomposition with the **group [i..j]** syntax, where *i* and *j* are the lower and upper bounds of the cardinality. When defining a cardinality, one can use the asterisk character * to denote the number of children in the group, for instance **group [1..*]** would be equivalent to **group someOf**. Optional features like *Accessories* are declared by putting the **opt** keyword in front of their name.

FMs most commonly have a tree structure but, sometimes, a directed acyclic graph (DAG) structure – a feature can have

several parents – might be useful [5]. DAG structures can also be modelled in TVL with the **shared** keyword associated to a feature name. It is illustrated in the following example where feature *D* has features *B* and *C* as parents:

```
root A
  group oneOf {
    B group allOf {D},
    C group allOf {shared D}
  }
```

B. Attributes

In our example, the *Motherboard* has four attributes: a price, a width, an height and a socket type. TVL supports four different attribute types: integer (**int**), real (**real**), Boolean (**bool**) and enumeration (**enum**). Furthermore, in our example, the *price* value is limited to values between 0 and 500. In TVL, this is expressed as follows:

```
Motherboard {
  int price in [0..500];
  int width;
  int height;
}
```

Attributes are thus declared by defining their type and name inside the definition block of the feature they belong to. Each attribute declaration is terminated by a semicolon. The **in** keyword is optional, it can be used to restrict the domain of an attribute. When declaring an attribute as an enumeration type, this means that it will take exactly one of a set of predefined values. The *socket*, for instance, is either *LGA1156* or *ASB1*.

```
Motherboard {
  enum socket in {LGA1156, ASB1};
}
```

For enumerations, the **in** keyword is mandatory. Notice the use of curly braces here as opposed to square brackets for the *price* attribute above. In TVL, square brackets are used to declare intervals and braces to declare lists.

In many cases, the value of an attribute will be calculated based on the values of some other attributes. The value of the *price* attribute of *Accessories*, for example, is the sum of the prices of its children *KeyboardAndMouse*, *PhilipsScreen* and *SamsungScreen*. Furthermore, the value of an attribute might also depend on whether its containing feature is selected or not. All this is written as follows in TVL:

```
Accessories {
  int price is sum(selectedChildren.price);
  group [0..2] {
    KeyboardAndMouse {
      int price is 19;
    },
    PhilipsScreen {
      int price is 99;
    },
    SamsungScreen {
      int price, ifIn: is 149, ifOut: is 0;
    }
  }
}
```

The keyword **is** can be used to set a fixed value for an attribute, e.g. *price* of *KeyboardAndMouse*. The keywords **ifIn**:

and **ifOut**: are guards that allow to specify the value of the attribute in the case in which the containing feature is selected (**ifIn**:) or not selected (**ifOut**:). We illustrate this with the *price* attribute of the *SamsungScreen* whose value will be 149 if the feature is selected and 0 if not.

While the price of the *KeyboardAndMouse*, *PhilipsScreen* and *SamsungScreen* features is fixed, the price of the *Accessories* is calculated: it is the sum (using the aggregation function **sum**) of the values of the *price* attribute of its selected children (using the **selectedChildren** keyword). Other operators are available and will be discussed in next section. A common modelling pattern for attributes declared for all features is to compute the value of the parent feature's attribute by aggregating the attribute values of its children, up to the root. The price of a *Computer*, for example, will be calculated by summing the prices of its selected sub-features, which in turn depend on the prices of their sub-features, and so on until leaf features with fixed price values are reached.

C. Expressions

In TVL, expressions are used to determine the value of an attribute as well as to express constraints on the FM. The language is strongly typed, each expression being either of type *bool*, *real* or *int*.

A basic expression is either an integer, a real, a Boolean, or a reference to a feature, an attribute or a constant. Those basic expressions can then be combined using classical operators: **+**, **-**, **/**, *****, **abs**, for numeric values; **!**, **&&**, **||**, **->**, **<->** for Boolean values as well as comparison operators **>**, **>=**, **<** or **<=**. Classical FM cross-tree constraints **excludes** and **requires** can also be used as Boolean expressions.

Furthermore, there are a number of aggregation functions **sum**, **mul** (multiplication), **min**, **max**, **avg** (average), **count**, **and**, **or** and **xor**. These aggregation functions can simply be used on lists of expressions or they can become powerful shorthand notations when used in combination with the **children** or the **selectedChildren** keywords. These allow to aggregate the value of an attribute that is declared for each child of a feature. The notation is **fct(children.attribute)**, or **fct(selectedChildren.attribute)** if the aggregate should be calculated on selected children only.

D. Constraints

Constraints in TVL are attached to features. They are simply Boolean expressions that can be added to the body of a feature definition. As with attribute declarations, they are terminated by a semicolon. The **ifIn**: and **ifOut**: guards we have previously seen can be used on constraints, too. In our example, the *socket* attribute of the *Motherboard* feature depends on the choice of the actual motherboard. One way to model this in TVL is to define a constraint in each child feature which basically 'sets' the value of its parent's attribute.

```
Motherboard {
  enum socket in {LGA1156, ASB1};
  group oneOf {
    Asus {
      ifIn: parent.socket == LGA1156;
```

```
    },
    Aopen {
      ifIn: parent.socket == ASB1;
    }
  }
}
```

E. Modularisation mechanisms

TVL offers various mechanisms that can help users to modularise large models. First of all, custom types can be defined at the top of the file and then be used in the FM. This allows to factor out recurring types. For instance, one might want to define the different sockets upfront and then use it as a type in an attribute declaration:

```
enum cpuSocket in {LGA1156, ASB1};
...
Motherboard {
  cpuSocket socket;
}
```

It is possible to define structured types to group attributes that are logically linked. A *dimension*, for instance, is a couple (height, width) and can be declared as such using a structured type. This type can then be reused inside the *Motherboard* feature:

```
struct dimension {
  int height;
  int width;
}
...
Motherboard {
  dimension size;
}
```

Users can also specify constants using the **const** keyword followed by a type, a name and a value. These constants can then be used inside expressions or cardinalities.

```
const int maxRamBlocks 4;
```

One can also use the **include** statement, which takes as parameter the path of a file (relative to the file containing the root feature). As expected, an **include** statement will include the contents of the referenced file at this point. Includes are in fact preprocessing directives and do not have any meaning beyond the fact that they are replaced by the referenced file.

```
include (./some/other/file);
```

Another mechanism is that features can be defined at one place and then extended further in the code. Basically, once a feature has been defined in the group block of its parent feature, its definition can be extended any number of times. In order to extend a feature definition, one just adds a feature block with the same name to the file. This block cannot be inside another feature, it has to start its own hierarchy. Each feature block may add constraints and attributes to the feature body. The children (with the **group** keyword) can only be defined in a single one of these blocks.

This mechanism allows modellers to organise the FM according to their preferences and can be used to implement separation of concerns [6]. For example, one could declare part of the structure of the FM without detailing each feature's attributes and instead provide them later on:

```

root Computer {
  group allOf {
    Motherboard,
    CPU,
    GraphicCard,
    opt Accessories
  }
}
Computer {
  int price is sum(selectedChildren.price);
}

```

In this example, the decomposition of the root is defined at the beginning while its attributes are declared further down. The advantage of this is that the structure is easily understandable because it is not cluttered by attribute declarations.

III. RELATED WORK

By far the most widely used notation in the literature is the graphical FM notation based on FODA [2]. Most of the subsequent proposals such as FeaturSEB [7], FORM [5] or Generative Programming [8] only slightly modify this graphical syntax (e.g. by adding boxes around feature names).

One exception is Batory [9] who proposed the GUIDSL syntax, in which the FM is represented with a grammar. The GUIDSL syntax is further used as a file format of the feature-oriented programming tools AHEAD [9] and FeatureIDE [10]. The GUIDSL format is aimed at the engineer and is thus easy to write, read and understand. However, it does not support arbitrary decomposition cardinalities, attributes, or the representation of the FM as a hierarchy.

Van Deursen and Klint [11] proposed the Feature Description Language (FDL), a textual language to describe features. FDL does not support attributes, cardinality-based decompositions, DAGs or duplicate feature names.

The SPLOT [12] and 4WhatReason [13] tools use the SXFM syntax and file format. While the format uses XML for metadata and the overall file structure, its representation of the FM is entirely text-based with the explicit goal to make it suitable for the engineer. It differs from the GUIDSL format in that it makes the tree structure of the FM explicit through (Python-style) indentation. It supports decomposition cardinalities but not attributes.

The feature modelling plugin [14] and the Fama framework [15] both use XML based file formats in which the whole FM is encoded in XML. These formats were not intended to be written or read by the engineer and are thus hard to interpret, mainly due to the overhead caused by XML tags and technical information that is extraneous to the model.

IV. CONCLUSION

We argue that while graphical FM languages may be more intuitive, they are not always adapted to large FMs involving attributes and complex constraints. We propose TVL, a text-based variability modelling language with a C-like syntax. The goal of the language is to be *scalable*, by being *concise* and by offering mechanisms for *modularity*. TVL is also meant to be *comprehensive* so as to cover a wide range of FM dialects proposed in the literature. We acknowledge that for

non IT stakeholders or for informal discussions around the blackboard, graphical FMs might be more appropriate than TVL. An advantage of text-based languages is that there are many well-accepted applications (viz. text editors, source control systems, diff tools, and so on) that support modelling and evolution out of the box. Furthermore, choosing a C-like syntax means lower learning curves for most software engineers. We hope that this will lead to an easier adoption of FMs in an industrial context.

At the moment, TVL is a language proposal, and we are requesting feedback from the variability modelling community. We developed a reference implementation for TVL in Java.² The library has two components; the *syntactic component* is a parser that performs type checking, checks well-formedness, and can normalize a model (eliminate syntactic sugar). Among other things, it can be used to implement TVL support in existing FM tools. The *semantic component* is able to translate a TVL file to either a Boolean CNF formula (if it does not contain numeric attributes), or to a CSP problem according to the formal TVL semantics defined in [4].

ACKNOWLEDGEMENTS

This work was partially funded by the Walloon Region, the Interuniversity Attraction Poles Programme, Belgian State, Belgian Science Policy (MoVES project), the BNB, the FNRS.

REFERENCES

- [1] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
- [2] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," SEI, CMU, Tech. Rep., 1990.
- [3] D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés, "Automated reasoning on feature models," in *Proceedings of CAiSE'05*, 2005.
- [4] A. Classen, Q. Boucher, P. Faber, and P. Heymans, "Syntax and semantics of TVL, a text-based feature modelling language," PRECISE Research Centre, Univ. of Namur, Tech. Rep., 2010.
- [5] K. C. Kang, S. Kim, J. Lee, K. Kim, G. J. Kim, and E. Shin, "Form: A feature-oriented reuse method with domain-specific reference architectures," *Ann. Softw. Eng.*, vol. 5, pp. 143–168, 1998.
- [6] P. Tarr, H. Ossher, W. Harrison, and S. M. J. Sutton, "N degrees of separation: multi-dimensional separation of concerns," in *ICSE'99*, 1999.
- [7] M. L. Griss, J. Favaro, and M. d. Alessandro, "Integrating feature modeling with the rseb," in *Proceedings of ICSR'98*, 1998.
- [8] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [9] D. S. Batory, "Feature Models, Grammars, and Propositional Formulas," in *Proceedings of SPLC'05*, 2005.
- [10] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel, "FeatureIDE: A tool framework for feature-oriented software development," in *Proceedings of ICSE'09*, 2009.
- [11] A. Deursen and P. Klint, "Domain-specific language design requires feature descriptions," *Journal of Computing and Information Technology*, vol. 10, p. 2002, 2002.
- [12] M. Mendonca, M. Branco, and D. Cowan, "S.P.L.O.T. - Software Product Lines Online Tools," in *Proceedings of OOPSLA'09*, 2009.
- [13] M. Mendonca, "Efficient reasoning techniques for large scale feature models," Ph.D. dissertation, University of Waterloo, 2009.
- [14] M. Antkiewicz and K. Czarnecki, "Featureplugin: Feature modeling plug-in for eclipse," in *Proceedings of the OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop*, 2004.
- [15] D. Benavides, S. Segura, P. Trinidad, and A. R. Cortés, "Fama: Tooling a framework for the automated analysis of feature models," in *Proceedings of VaMoS'07*, 2007.

²See the TVL website at <http://www.info.fundp.ac.be/~acs/tvl>.

XToF – A Tool for Tag-based Product Line Implementation

Christophe Gauthier, Andreas Classen,*
 Quentin Boucher, Patrick Heymans

PRECISE Research Centre
 Namur, Belgium

Email: chistophe.gauthier@student.fundp.ac.be
 {acs,qbo,phe}@info.fundp.ac.be

Margaret-Anne Storey

University of Victoria
 Victoria, Canada

Email: mstorey@uvic.ca

Marcílio Mendonça

University of Waterloo
 Waterloo, Canada

Email: marcilio@csg.uwaterloo.ca

Abstract—This tool demo paper describes a tool called XToF which is being developed through a collaboration between the University of Victoria, the University of Namur and the University of Waterloo. The purpose of the tool is to let programmers define, maintain, visualise and exploit precise traceability links between a feature diagram and the code base of a software product line. The resulting tool supports automated configuration of a Java or C code base and is minimally intrusive with respect to development practices.

Index Terms—tool demo; software product line; feature diagram; tagging; programming language; C; Java

I. INTRODUCTION

The tool described in this paper is being developed as part of the Masters thesis of the first author. The tool extends a toolchain that was previously assembled/developed by the University of Namur and Spacebel, a Belgian company that develops software for space missions.

The purpose of both tools (the old and the new one) is to let programmers define, maintain, visualise and exploit precise traceability links between a feature diagram (FD) and the code base of a software product line. Both tools are meant to be minimally intrusive with respect to development practices. The new tool, called XToF¹, provides enhanced functionality by leveraging on two new components: (1) TagSEA, an Eclipse plug-in developed at University of Victoria, which purpose is to support navigation and knowledge sharing in collaborative program development, and (2) S.P.L.A.R. a Java library developed at University of Waterloo that automates various FD analyses.

The remainder of the paper is structured as follows. It starts in Section II with a description of the requirements (Section II-A) and implementation (Sections II-B and II-C) of the initial toolchain, together with a list of its limitations (Section II-D). Then, in Section III, we present the contribution of this paper: XToF, the new prototype designed to overcome the aforementioned limitations. We describe in turn its components and principles (Section III-A), its functionalities (Section III-B) and on-going as well as future development (Section III-C).

*FNRS Research Fellow.

¹XToF stands for cross(X)-Tagging of Features

II. THE INITIAL TOOLCHAIN

A. Context and requirements

The assembly/development of the initial toolchain took place as a collaboration between the University of Namur and Spacebel. The goal of this collaboration was to turn the implementation of a flight grade satellite communication software library into a software product line that would support the following requirements:

- allow *mass-customisation* of the library: meaning to be able to efficiently derive products that only contain the features required for a specific space mission,
- *be compliant with quality standards and regulations* in place for flight software,
- have a *minimal impact on current development practices*,
- *automate* the solution as much as possible.

The first and second requirements stem from the strict constraints that are imposed on flight grade on-board software. Components for space usage are developed to deal with extreme environmental conditions such as cosmic rays, temperature variation and vibration. This type of hardware is usually very expensive and several evolutionary steps behind the consumer hardware we more commonly know. Therefore, CPU usage and memory footprint typically have to be minimised. Also, developers often have no other choice than programming in C and obey strict rules that prohibit usage of ‘dangerous’ mechanisms, such as dynamic heap memory allocation, or general-purpose third-party libraries [1]. Along the same lines, *dead code* is also to be avoided. In our case, since specific missions only require part of the protocol’s functionality, it is important to only deploy those parts (or features) of the protocol that are going to be used.

The rationale for the third and fourth requirements was to facilitate adoption of the solution by the company. A first version of the toolchain was then elaborated jointly by the academic and industry partners. It is described in detail elsewhere [2]. In the rest of this section, we just recall its most important features and limitations to introduce our new contribution, XToF, presented in Section III.

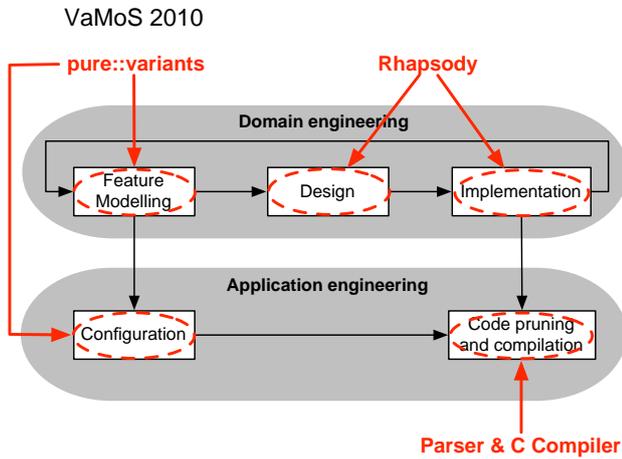


Fig. 1. Toolchain as deployed at Spacebel

B. Deployed toolchain

Guided by the above requirements, a first version of the process and toolchain were developed and successfully deployed in the company. The tool-supported process is depicted in Figure 1. It is organised after the classical software product line engineering process [3] which consists of two main streams: *domain engineering* (the creation of reusable artefacts or core assets) and *application engineering* (the usage and adaption of the core assets to create final products). In our case, the core assets are the Feature Diagram (FD), the system architecture (in UML) and the C code base, made of 6224 lines of code, the two last assets being decorated with tags pointing to the features of the FD. Application engineering starts with a configuration step during which some features are selected and some are discarded. This information is then used to remove code related to discarded features before the mission-specific product is compiled. A technical report [2] describes each of these steps in greater detail. It also gives an extensive definition of the syntax and semantics of our tagging language, demonstrates its correctness, provides an illustration and compares it with other annotative approaches to software product line implementation such as CIDE [4] and `#ifdef` pre-processing statements.

Figure 1 also shows how this process is supported by tools:

- design and implementation are supported by the tools already in use at the company, namely the *Rhapsody* UML CASE tool, and the C compiler;
- feature modelling and configuration are supported by *pure::variants*, a commercial off-the-shelf FD-based tool;
- a parser that was developed specifically for this project. The parser takes two inputs: (1) a valid list of features provided by *pure::variants* after configuration, and (2) an ANSI C source file annotated with tags written in our tagging language. It returns an ANSI C source file with no tags and no unnecessary code. The parser is encapsulated in a make-file and run on every single file of the codebase.

C. The tagging languages

Basically, a feature tag is an annotation of a block of C code with the names of the features that require the block to be present. If none of the features listed in a tag is included in a particular product, then the tagged code block will not be part of the source code generated for this product. Tags can be nested and a whole file can be tagged with a special annotation. Untagged code is assumed to be needed for all features.

Syntactically, a feature tag is a comment that follows a predefined pattern. As such, it is displayed in the same colour as comments in code editors. The syntax of feature tags is:

```
<fcomment> ::= "/*@feature:" <flist> "@*/" [<filetag>]
<flist> ::= <featurename> ( ":" <flist> ) *
<filetag> ::= "/*@!file_feature!@*/"
```

where `<featurename>` identifies a feature of the FD. The scope of a tag is the functional block, which we define as a group of statements that belong together, and that can be removed as a whole without violating the syntax or grammar of the language. For instance, it would be impossible to remove only the signature of a function without also removing its body. Functional blocks thus correspond to elements of the abstract syntax tree (AST), an idea previously found in [4]. With this approach, we can guarantee that the pruned code will always be syntactically correct. The functional block corresponding to a code tag is determined by the instructions that follow the tag. More details can be found in the technical report [2].

D. Limitations of the toolchain

The tool-supported process described in the previous sections turned out to be effective in meeting the requirements set out by the company. A detailed evaluation [2] revealed that there was still space for improvements (in order of priority):

- *Tighter integration*: communication between the tools was performed only through file exchange. Although this did not impede usage of the toolchain, it was recognised that an integrated environment, where loosely coupled tools play together, could be a significant enhancement. An important improvement, for example, could be that the feature editor/configurator could point directly to the code fragments a feature corresponds to in the code editor, and vice versa.
- *Legibility*: according to the company's developers, the legibility of the source code was not reduced by the tags. Indeed, the tagging language was designed to be concise and is rendered in a different colour in most code editors. However, the developers found it sometimes hard to determine the feature(s) corresponding to a specific source fragment, especially in the presence of nested tags. Tag-based filtering and visualisation techniques could alleviate this problem.
- *Portability*: although pruning dead code is most usually required in embedded systems where C dominates, C is not the only language used in embedded systems. Additionally, our "tag and prune" approach has a wider

applicability than embedded systems, hence the idea of extending the approach to other languages.

- *On-the-fly tag generation*: the programmers who used the toolchain estimated that the overhead due to the tags during the domain implementation phase was 20 to 25% with respect to tag-free implementation of a ‘maximal’ product (the return on this investment being delayed to the application implementation phase). However, they also recognised that the overhead could be decreased if the tags were systematically captured at the time the feature is programmed rather than after the fact.

III. THE NEW PROTOTYPE: XTOF

Functionally, XToF, the new prototype, is meant to support the activities depicted in Figure 1 in a single integrated environment, and overcome the limitations described in the previous section.

A. Components and principles of XToF

The opportunity for re-implementing the original toolchain came from the discovery of an open-source Eclipse plug-in called TagSEA. TagSEA was developed by Storey *et al.* [5] to support asynchronous and collaborative program development. It enhances navigation and knowledge distribution in the code based on tags placed by the programmers. The approach and tool are originally unrelated to software product lines, but turned out to be applicable in this context.

XToF uses the capabilities of TagSEA to manage tagging and tags. TagSEA defines *waypoints* as “locations of software model elements”[6]. The notion of waypoint as a *point of interest* has been extended to a *design area* of interest in order to capture blocks of code associated to feature tags. TagSEA provides mechanisms to filter tags, list waypoints and navigate to a waypoint. XToF then links TagSEA waypoints to features and blocks of code.

One of the main enhancements to the first toolchain is that the FD is now displayed directly in the programming environment. The FD is used as an index to code fragments and as the configuration interface. One can select a set of features to obtain specific views of the program and to configure it. XToF adopts the classical layout of Eclipse (see Figure 2): the FD is displayed as a directory tree (A), some buttons (B) trigger actions like configuration or tag filtering, while TagSEA constantly displays the list of waypoints (D) for each tag (C).

To allow tighter integration of TagSEA with the FD-related functionalities, we needed full access to their source code. This was provided by SPLAR² a powerful Java library that automates various FD analyses, by which we replaced *pure::variants*.

B. Current functionalities

We now take a closer look at the tool’s currently supported functionalities:

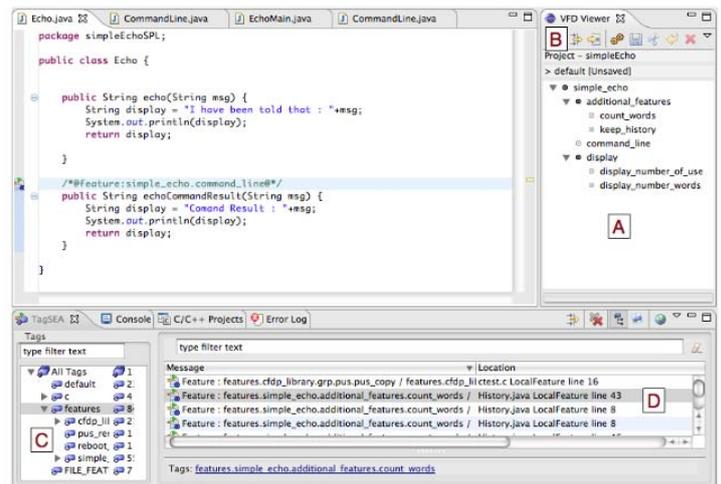


Fig. 2. XToF’s main screen

- *Loading the FD*: To be displayed and configured in the tool, the FD has to be loaded. XToF expects it as an XML file in the *SXFM* format.³ The file can be created in any text editor, but can be more easily produced by the web-based visual FD editor SPLLOT [7], the front-end to SPLAR. Once the FD is loaded, XToF displays it and lets the users add tags, navigate and configure. The loaded FD is copied to the project folder and its path is saved as a property of the project. The FD is thus made available to all project contributors who can work in parallel.
- *Tagging code fragments*: To reduce the time needed to tag blocks of source code, XToF uses auto-completion from Eclipse. While typing a tag, feature names are displayed, and when selected, directly added to the tag.
- *Navigation and visualization*: XToF feature tags behave like regular TagSEA waypoints. The user can list the location of feature tags, navigate to a tagged code fragment and display it. Some visualisations have been developed to answer simple questions such as “Which blocks are associated to a set of tags?” and “Which set of tags is associated to a line of source code?”. To answer the first question, the user can select the set of tags in XToF and the tagged block of source code is highlighted. Another mechanism provides the opposite function, i.e. answers the second question: the features corresponding to the current line in the active editor window are highlighted in the FD. Additionally, XToF filters packages and classes that contain blocks tagged with selected features from the FD. Finally, we reuse a ‘cloud’ visualization from TagSEA that shows how tags are used.
- *Configuring and pruning*: Configuration and pruning are now integrated. The configuration interface is based on the FD. Clicking on a feature allows the user to toggle it from deselected to selected and conversely (see Figure

²See <http://www.splot-research.org>

³See <http://gdansk.uwaterloo.ca:8088/SPLLOT/sxfrm.html>

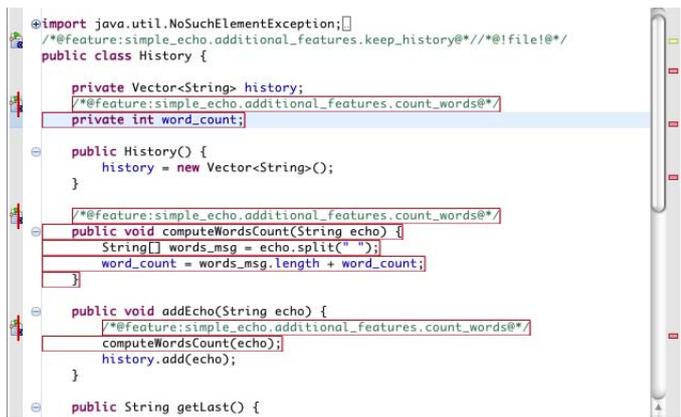


Fig. 3. Code highlighting in XToF

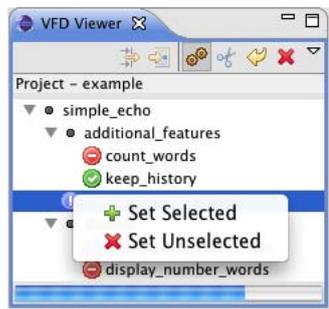


Fig. 4. Product configuration in XToF

4). Each decision made on the diagram is propagated by SPLAR to ensure the validity of the configuration. Once configuration is completed, the mission-specific implementation can be generated. To do this, XToF will clone the project to a new one with a name provided by the user. It will then prune the source code and remove code according to the valid list of features.

- *Portability*: XToF takes advantage of the plug-in platform provided by Eclipse to support other languages than Java. Two languages are currently supported: Java and C.

C. On-going and future work

Additional functionalities will be implemented in the future:

- *Editing the FD*: The current version of XToF does not support editing of the FD. The objective would not only be to create it from the same front-end as the rest of the functionalities, but also maintain the consistency between the FD and the feature tags. For example, if one modifies the name of a feature, each tag that uses the feature will be modified too, thereby supporting co-evolution of the FD and the program.
- *High level visualization*: The current visualization is limited. It is hard to determine which files and packages are tagged with given features (although not impossible if one reads the list of tagged blocks). XToF will provide high level visualization to answer the questions: “Which files and packages are associated to a set of features and

conversely?” and “Which packages or classes (in the case of an OO project) have features in common?” XToF will display links between the files (or classes), packages and features, and provide mechanisms to restrict the view to a set of features or files.

- *Improve declaration tagging*: When the partner company used the first prototype, they reported that one of the major sources of errors was incorrect tagging of variable, function and type declarations [2]. This can occur, for example, when a variable tagged with feature *A* is required by a feature *B* without the developer updating the tag. The product generated with feature *B* selected but feature *A* deselected will not compile. To prevent such an error, the ‘using’ feature must declare the variable, or the ‘declaring’ feature must always be present with the ‘using’ feature. XToF will help the user avoid such issues by offering a pruning based on the set of features that are always present with one selected feature [2]. Another possibility that we will investigate is to take into account dependencies in the code for automatic generation of tags, in the spirit of change-oriented programming [8].

IV. CONCLUSION

In this paper, we introduced XToF, a tool prototype supporting tag-based product line implementation in Java and C. XToF is an extension of a toolchain that was initially developed as joint university-industry project and which has been deployed in the company. XToF will supersede this toolchain by improving it in various ways: better tool integration, visualization, portability to other programming languages and on-the-fly tag generation. Once finished, XToF will provide an integrated tool to create a feature diagram, develop a tagged ‘maximal’ product, navigate through the features and the tagged blocks of code, classes and packages (in the case of OO programs), support feature-based configuration and generate products automatically by pruning the source code.

V. ACKNOWLEDGEMENTS

This work is sponsored by the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy (MoVES project), FEDER, BNB and FNRS.

REFERENCES

- [1] MISRA, *MISRA-C: Guidelines for the use of the C language in critical systems*. Motor Industry Research Association, 2008.
- [2] Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, and L. Demonceau, “Tag and prune: A pragmatic approach to software product line implementation,” PRECISE Research Centre, Univ. of Namur, Tech. Rep., 2009.
- [3] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
- [4] C. Kästner, S. Apel, and M. Kuhlemann, “Granularity in software product lines,” in *Proceedings of ICSE '08*, 2008.
- [5] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby, “Shared waypoints and social tagging to support collaboration in software development,” in *Proceedings of CSCW '06*, 2006.
- [6] —, “Waypointing and social tagging to support program navigation,” in *In proceedings of CHI '06*, 2006.
- [7] M. Mendonca, M. Branco, and D. Cowan, “S.P.L.O.T.: Software product lines online tools,” in *Proceeding of OOPSLA'09*, 2009.
- [8] P. Ebraert, A. Classen, P. Heymans, and T. D'Hondt, “Feature diagrams for change-oriented programming,” in *Proceedings of ICFI'09*, 2009.

Tool Support for Evolution of Product Lines through Rapid Feedback from Application Engineering

Wolfgang Heider Rick Rabiser

Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University Linz, Austria
{heider | rabiser}@ase.jku.at

Abstract—Product lines are maintained and evolved over many years. Technology changes, internal enhancements, and customer requests lead to new requirements and adaptations of existing capabilities. New requirements are important as they drive the evolution of the product line. However, if multiple application engineering projects are conducted concurrently, managing new requirements can quickly become a tedious task. In particular, there is a lack of tools supporting the feedback loop from application engineering to domain engineering. In this paper we propose a tool-supported approach for managing application requirements. The approach aims at supporting tracking of requirements in multiple application engineering projects to foster the definition of new domain requirements.

Keywords – reactive product line evolution; requirements management.

I. INTRODUCTION

Product lines are developed and used for many years and need to be continuously evolved to remain successful. Managing evolution is therefore success-critical for any product line approach. Engineers need to consider new requirements which lead to changes and extensions to the product line's assets and the derived products [1]. Some requirements might only be relevant for specific products while others might affect the product line as a whole [2][3]. Distinguishing between application and domain requirements is thus essential for requirements management in product line engineering. In the simple case, new requirements might be addressed just by providing new choices in a variability model. Often, however, the situation is more challenging. New requirements might trigger ideas for products in new domains. They might also have an impact on the product line architecture and result in significant refactoring. Process and tool support is thus needed for effective application requirements management and the extraction of domain requirements from application requirements [4][5].

In this paper we describe our ongoing research on monitoring and tracking requirements emerging in concurrent application engineering projects and tool support we have been developing for this purpose. Our main aim is to accelerate the innovation cycle in product lines by supporting reactive product line evolution. Reactive evolution means that the product line is iteratively extended after each application engineering project by considering the newly collected applications requirements as potential new domain requirements.

Reactive evolution has already proven successful to support the transition from single-system engineering to product line engineering [6]. We do not disregard the important role of proactive product line evolution for innovation [7][8] but intentionally focus on reactive product line evolution in this paper. Referring to a classification of the "scope of reuse" of systems as presented by Deelstra *et al.* [9], our approach focuses on software product lines and configurable product families.

The remainder of the paper is structured as follows: In Section II we discuss practical challenges of reactive product line evolution. In Section III we outline our approach of a continuous innovation cycle that allows fast feedback from application to domain engineering. In Section IV we briefly introduce the DOPLER product line engineering tool suite [10], discuss its features for application requirements management, and describe tool support for tracking and gathering requirements in concurrent application engineering projects. We conclude the paper with a short discussion of issues and an outlook on future work.

II. REACTIVE PRODUCT LINE EVOLUTION IN MULTI-PROJECT ENVIRONMENTS

Each application engineering project potentially contributes to the evolution of the product line by giving rise to new requirements and innovative ideas. In industrial settings it is common that multiple products are derived concurrently from one product line. This leads to diverse requirements affecting the product line and its variability (e.g., additional functionality or modifications of existing components). Apparently, these new requirements should not directly lead to modifications of the product line. Instead, the new requirements have to be managed and their impact has to be analyzed. Some requests might be best addressed with product-specific development. However, there is the danger that engineers do not provide adequate and timely feedback to domain engineering due to the constant pressure in projects. This means that the product line does not evolve as much as it could.

Figure 1 depicts a typical application engineering process in a multi-project environment and shows how feedback to domain engineering usually is managed. The (simplified) application engineering phase comprises four steps: (1) A product derivation project is initiated for a new product being derived for e.g., a new customer. An initial product is derived

based on the product line's existing assets that best match the requirements of the customer. (2) The sales person or analyst negotiating with the customer captures new customer requirements that cannot be fulfilled with the existing assets of the product line. (3) Application engineers address the new requirements by developing product-specific extensions or by adapting and reconfiguring existing components. (4) The product is delivered to the customer.

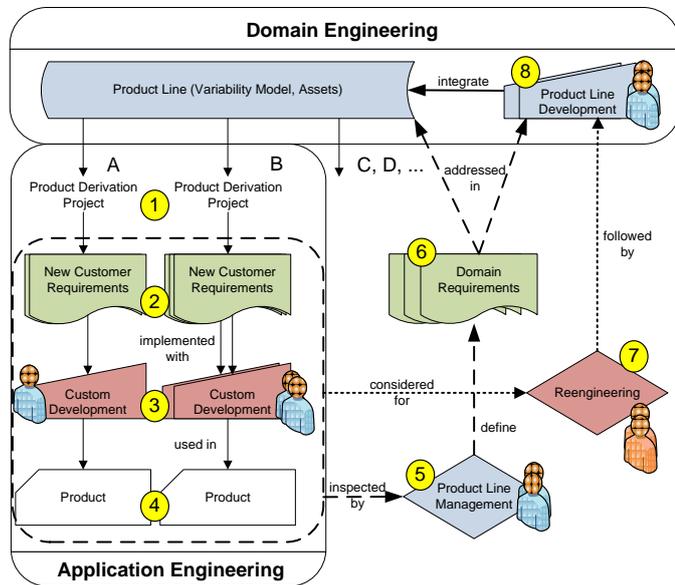


Figure 1. A typical process for reactive product line evolution: Delayed feedback is provided to domain engineering after application engineering projects are finished. This approach often leads to costly reengineering of project-specific developments and impedes the evolution of the product line.

In such a process the evolution phase comprises four steps: (5) Product line managers analyze the requirements and identify potential candidates for domain requirements. (6) Product line engineers define domain requirements based on the selected application requirements by considering the existing product line and its variability. (7) At the same time core asset developers analyze product-specific components regarding their suitability for reengineering and inclusion in the product line. (8) Product line engineers extend product line variability models and adapt the product line architecture based on the results of steps 6 and 7.

Although intuitive and straightforward this process has several drawbacks:

- Changes to the derived product have to be maintained after the product has been delivered to the customer. The customer-specific developments might differ so much from the initially derived product that the benefits of pursuing a product line approach are diminished.
- It is hard to decide post-hoc, which of the changes and additional developments made in multiple application engineering projects can be included in the product line. The lack of traceability from customer-specific requirements and product-specific solutions to domain requirements further complicates this issue.
- Often similar extensions and adaptations are made independently of each other in concurrent application engineering projects. This can lead to unnecessary effort and violates the

principles of reuse. It is also hard to identify similar changes post-hoc due to different implementation solutions [3].

- Because of the time pressure in application engineering projects "quick-and-dirty" solutions are likely to be pursued. In many cases this leads to a high effort for redevelopment when integrating product-specific developments into the product line [5][7].

In response to these problems we propose an approach to better coordinate requirements management in concurrent application engineering projects. Our goal is to accelerate product line evolution and innovation by fostering a short feedback cycle to domain engineering activities. Our aim is to increase the reuse rate for development and to reduce custom development.

III. FAST FEEDBACK FROM APPLICATION ENGINEERING TO DOMAIN ENGINEERING

New customer requirements captured during application engineering are a driver for evolution and a source of innovation in product line engineering. If the development performed for one specific customer is potentially interesting and useful for other customers it should be integrated in the product line as soon as possible [5]. We therefore propose a rapid feedback loop from application engineering to domain engineering as shown in Figure 2:

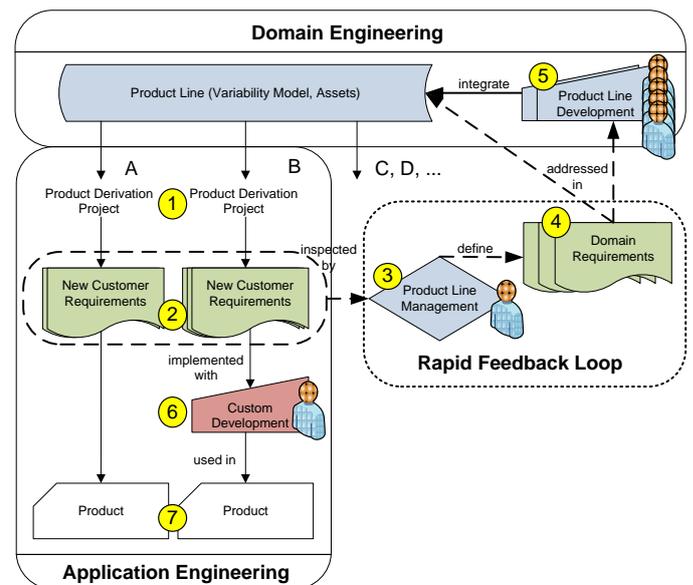


Figure 2. A rapid feedback process for reactive product line evolution: Early and rapid feedback is provided to domain engineering by tracking and communicating new and changed requirements in concurrent application engineering projects..

- (1) A project is initiated to derive a product from the product line that suits the requirements of the project and customer as far as possible. (2) A sales person or project manager captures new customer requirements during product derivation that cannot be satisfied with the product line. Engineers define relations of newly captured requirements with existing product line assets and variation points. Rapid feedback already triggers evolution in this step: (3) Product line management analyzes the captured requirements to identify potential domain requirement candidates. Changes to requirements in concurrent application engineering projects are tracked and candidates for domain requirements are identified by aggregating and/or splitting requirements gathered in

the projects. (4) The product line engineers select new domain requirements from the candidate requirements. Ideally, the new requirements are already linked to existing product line assets and variation points (cf. step 2) to streamline the planning of development. (5) Product line engineers address the domain requirements by implementing additional components and adapting variability models. (6) After integrating the product derived earlier with newly developed product line components, application engineers perform custom development and product customization for requirements that can and/or shall not be integrated into the product line. (7) The product is delivered to the customer.

IV. TOOL SUPPORT FOR RAPID FEEDBACK

This process aiming at early and rapid feedback requires tool support. We have been extending our DOPLER product line engineering tool suite [10] for this purpose. DOPLER is based on decision-oriented variability models and supports both domain engineering and application engineering activities [11][12]. Product derivation (cf. step 1) with the DOPLER tools is based on questionnaires. A project manager answers questions thereby taking decisions and resolving variability. To support the rapid feedback cycle described above we rely on DOPLER's existing features for managing application requirements (cf. step 2) and on newly developed features for tracking the evolution in concurrent application engineering projects (cf. step 3).

A. Application Requirements Management Support

The DOPLER derivation tools already allow capturing and managing new customer requirements. A user answering questions can capture newly arising requirements together with typical attributes (e.g., description, rationale, risk level, priority, etc.). Traceability is established to other relevant artifacts: A new requirement is automatically related with its originating decision(s). It can also be related with assets that led to the requirement or assets that might need to be changed when realizing the requirement [12].

However, this existing support for managing application requirements and tracking their state is not sufficient to provide rapid feedback from application to domain engineering. We have thus been developing tool support for tracking the creation of new and modifications to existing requirements in multiple projects. This provides product line engineers with a cross-project view of all requirements to ease the planning and management of product line evolution.

B. EvoKing: Tool Support for Tracking Changes to Requirements in Concurrent Projects

We have been developing EvoKing [13], a tool that supports tracking the evolution of resources in Eclipse workspaces used by the DOPLER tools. EvoKing can easily be configured to recognize changes to arbitrary files, models, and model elements. For example, EvoKing can track changes to requirements in concurrent application engineering projects. Each time a new requirement is captured in a derivation project, EvoKing stores information about the change and establishes traceability to related development artifacts.

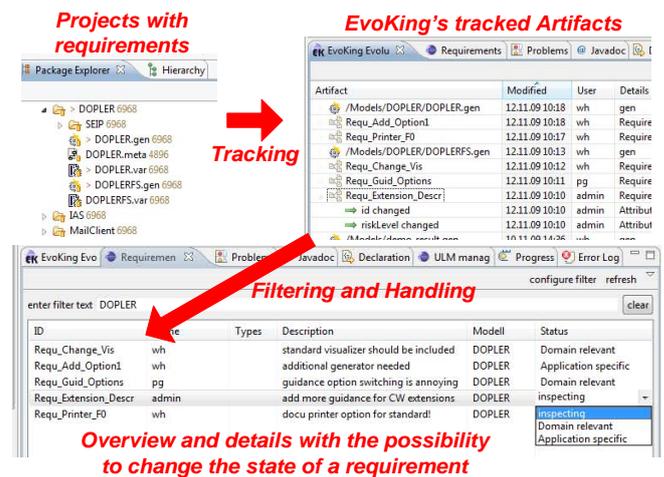


Figure 3. EvoKing tracks requirements in application engineering projects and provides an overview for planning subsequent product line evolution.

The upper right pane in Figure 3 shows a view of changes to DOPLER workspace elements as tracked by EvoKing. It visualizes relations between the artifacts in the workspace as well as changes made to the artifacts (e.g., requirements created or modified). The lower pane in Figure 3 shows a requirements view that gives an overview of all features, variations, and adaptations requested in all concurrent projects that are monitored by the tool. A product line engineer can mark candidate requirements for evolving the product line.

Figure 4 depicts a simple model showing states of requirements we use in our current implementation to support the rapid feedback loop from application engineering to domain engineering.

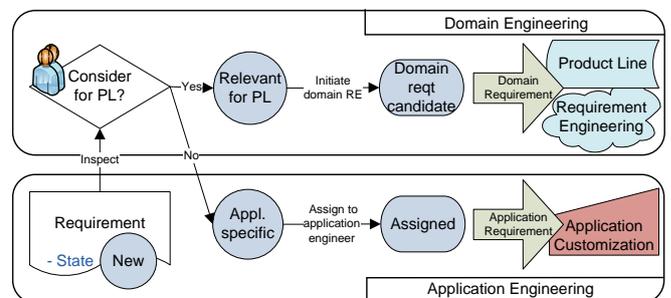


Figure 4. Selecting application requirements as domain requirement candidates.

A new requirement captured in an application engineering project is inspected by a product line engineer to assess the requirement's relevance for the product line. If relevant, the requirement is elaborated into a domain requirement candidate. If assessed as application-specific only, the requirement is assigned to developers in charge of custom development. Domain requirements can be managed in requirements management tools as described in [5]. Processes for handling and implementing domain requirements are, e.g., described by Pohl *et al.* [4] or Moon *et al.* [14].

V. CONCLUSIONS AND FUTURE WORK

Managing the evolution of a product line relies on highly sensitive and well-advised product line management. In this paper we discussed the need of providing rapid feedback from application engineering to domain engineering. Supporting such reactive evolution relies on flexible tool support for managing product line requirements. In this paper we have described tool support for collecting and tracking customer requirements in concurrent application engineering projects.

We believe that avoiding product-specific development as far as possible is a key to the success of a product line approach in practice. The main aim of our approach is to help avoiding product-specific development by facilitating fast feedback from application to domain engineering. However, our approach bears risk of delaying application engineering due to the overhead of domain requirements engineering and because of the extensive synchronization between application and domain engineers. Also, in practice some application requirements simply cannot or should not become domain requirements for various reasons. Fully avoiding product-specific development will therefore never be possible [9].

In future work we will explore the use of the DOPLER tool suite for domain requirements management by integrating domain requirements into the variability models. We will investigate the usage and integration of existing requirements engineering tools and the adaptation of already mentioned processes [4][14] for domain requirements engineering. We will also evaluate and validate our rapid feedback approach together with our industry partner.

ACKNOWLEDGMENT

This work has been conducted in cooperation with Siemens VAI Metals Technologies and has been supported by the Christian Doppler Forschungsgesellschaft, Austria.

REFERENCES

- [1] D. Dhungana, T. Neumayer, P. Grünbacher, and R. Rabiser, "Supporting Evolution in Model-based Product Line Engineering," Proc. of the *12th International Software Product Line Conference (SPLC 2008)*, Limerick, Ireland, IEEE Computer Society, 2008, pp. 319-328.
- [2] M. Svahnberg, and J. Bosch, "Evolution in software product lines: Two cases," *Journal of Software Maintenance*, vol. 11, no. 6, pp. 391-422, 1999.
- [3] K. Schmid, and M. Verlage, "The Economic Impact of Product Line Adoption and Evolution," *IEEE Software*, vol. 19, no. 4, pp. 50-57, 2002.
- [4] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*, Springer, 2005.
- [5] C. Kuloor, and A. Eberlein, "Requirements Engineering for Software Product Lines," *15th International Conference Software & Systems Engineering & their Applications (ICSEA2002)*, Paris, France, 2002.
- [6] R. Buhrdorf, D. Churchett, and C. W. Krueger, "Salion's Experience with a Reactive Software Product Line Approach," *PFE 2003*, pp. 317-322.
- [7] P. Knauber, "Managing the Evolution of Software Product Lines," *8th International Conference on Software Reuse (ICSR-8)*, Madrid, Spain, Springer LNCS, 2004.
- [8] K. Villela; J. Doerr, A. Gross, "Proactively Managing the Evolution of Embedded System Requirements," *International Requirements Engineering (RE'08)*, 2008, pp. 13 – 22.
- [9] S. Deelstra, M. Sinnema, and J. Bosch, "Product derivation in software product families: a case study," *Journal of Systems and Software*, vol. 74, no. 2, Jan., 2005, pp. 173-194.
- [10] D. Dhungana, P. Grünbacher, and R. Rabiser, "Domain-specific Adaptations of Product Line Variability Modeling," Proc. of the *IFIP WG 8.1 Working Conference on Situational Method Engineering: Fundamentals and Experiences*, Geneva, Switzerland, International Federation for Information Processing, Springer Series in Computer Science, 2007, pp. 238-251.
- [11] R. Rabiser, P. Grünbacher, and D. Dhungana, "Supporting Product Derivation by Adapting and Augmenting Variability Models," Proc. of the *11th International Software Product Line Conference (SPLC 2007)*, Kyoto, Japan, IEEE Computer Society, 2007, pp. 141-150.
- [12] R. Rabiser and D. Dhungana, "Integrated Support for Product Configuration and Requirements Engineering in Product Derivation," Proc. of the *33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'07)*, Lübeck, Germany, IEEE Computer Society, 2007, pp. 219-228.
- [13] W. Heider, R. Rabiser, D. Dhungana, and P. Grünbacher, "Tracking Evolution in Model-based Product Lines," *1st International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2009)*, Proc. (vol 2) of the *13th International Software Product Line Conference (SPLC 2009)*, San Francisco, USA, Software Engineering Institute, Carnegie Mellon, 2009, pp. 59-63.
- [14] M. Moon, K. Yeom, and Heung Seok Chae, "An Approach to Developing Domain Requirements as a Core Asset Based on Commonality and Variability Analysis in a Product Line," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, July, 2005, pp. 551-569.

Tool Support for Incremental Consistency Checking on Variability Models

Michael Vierhauser¹ Deepak Dhungana² Wolfgang Heider¹ Rick Rabiser¹ Alexander Egyed³

¹ Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University
Linz, Austria
rabiser@ase.jku.at

² Lero - The Irish Software Engineering Research Centre
University of Limerick
Limerick, Ireland
deepak.dhungana@lero.ie

³ Institute for Systems Engineering and Automation
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

Abstract—The complexity of variability models makes it hard for product line engineers to maintain their consistency over time. Engineers need support to detect and resolve inconsistencies. In this paper, we describe our initial results towards tool support for incremental consistency checking on variability models. The main aim of our research is to improve the overall performance and scalability of consistency checking. We report on experiences of integrating an existing incremental consistency checker in the DOPLER product line tool suite.

Keywords – variability models, incremental consistency checking, tool support.

I. INTRODUCTION AND MOTIVATION

Product line variability models are inherently complex. Independent of the modeling approach used (e.g., feature-oriented [1], decision-oriented [2], orthogonal [3]) real-world variability models can easily contain several thousand elements with diverse and often complex dependencies. Through the collaboration with our industry partner Siemens VAI – the world's leading company in engineering and plant-building for the iron, steel, and aluminum industries – we have learned that engineers in practice face big challenges in maintaining the consistency of variability models. The consistency of models is, however, essential for deriving correct products. It is also critical that variability models correctly reflect the actual system (e.g., components defined in the variability model must really exist). Therefore engineers should be supported in detecting and keeping track of inconsistencies during modeling.

Several consistency checking mechanisms have been reported in the literature and have been applied to various types of models [4], [5], [6]. A drawback of many of these approaches is that they are only capable of checking the consistency of entire models in a batch-oriented manner. This means that the consistency constraints are evaluated for the entire model at certain points in time (e.g., when saving a model). Due to the complexity of real-world models (often containing thousands of model elements and non-trivial dependencies), such a "batch-oriented" approach to consistency checking leads to performance problems. We also experimented with a batch-oriented approach in the context of our

DOPLER product line engineering tool suite [7]. To improve performance, we however decided to incorporate an existing approach for incremental consistency checking of UML design models [8] in the context of product line variability modeling. In this paper, we describe our initial results towards tool support for incremental consistency checking on variability models.

II. CONSISTENCY CHECKING FOR DOPLER VARIABILITY MODELS

In collaboration with Siemens VAI we have been developing the decision-oriented product line engineering approach DOPLER [9].

A. DOPLER modeling language

DOPLER variability models comprise two elements: *Assets* and *Decisions*. Assets represent the core elements in the product line (e.g., components). Assets can depend on each other functionally (e.g., one component requires another component) or structurally (e.g., a component is part of a sub-system). DOPLER allows modeling assets at an arbitrary granularity and with arbitrary attributes and dependencies (based on a given set of basic types). Users can create domain-specific meta-models to define the types of assets, their attributes, and dependencies [9].

In case of Siemens VAI, the asset types that are part of variability models are components (representing Spring [10] XML component descriptions which in turn represent Java Beans), properties (key-value settings), resources (e.g., configuration files), as well as documents (e.g., user documentation). Diverse domain-specific dependencies have been defined, for example, a component can *require* other components, a component can *require* properties, or a document can *contribute to* a resource.

In DOPLER variation points are represented with *decisions*. Decisions have a unique name and a question that is asked to a user during product derivation. They can depend on each other hierarchically (if a decision needs to be taken before another decision becomes "visible") or logically (if taking a decision changes the value of another decision). Possible types of decisions are Boolean, enumeration, string, and number.

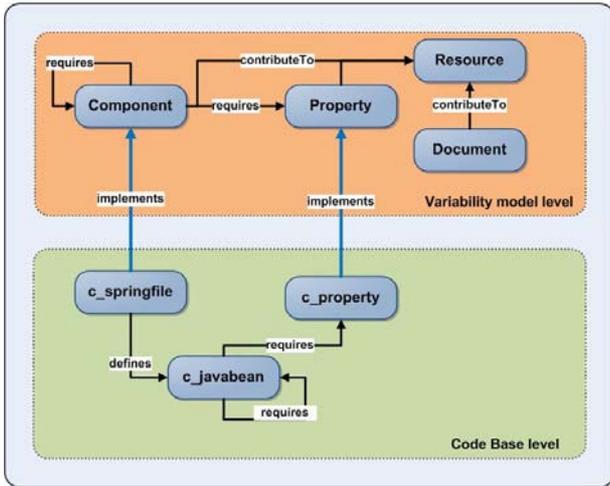


Figure 1. Siemens VAI meta-model overview. The upper part depicts the high-level meta-model. The lower part depicts additional elements needed to represent the code base of the product line. The relation of model level and file level is provided via the implements relation.

B. Consistency Constraints

When defining consistency constraints for Siemens VAI, our goal was to check consistency within the variability model as well as between the model and the code base of the product line. Our constraints therefore also check whether the model elements are consistent with concrete implementation artifacts like (Spring XML) component definitions and Java Beans. For constraints between the model level and the actual code base we generate a model representation of the code. For that reason the original DOPLER meta-model for Siemens VAI has been extended to cover information on the Spring files, the contained Java Beans, their properties, and the relations among the diverse elements (cf. Figure 1). Table 1 shows some examples of constraints that are relevant in the context of our industry partner.

We differentiate between generic and domain-specific (Siemens VAI) constraints. The generic constraints are relevant in any DOPLER variability model. For example, it is important to detect cycles between decisions (either based on hierarchical or logical dependencies among them) in any DOPLER model. These constraints are independent of the domain-specific meta-model (depicted in Figure 1). We therefore reuse these constraints and provide them as a core functionality of the consistency checker.

Siemens VAI-specific constraints mainly address model to code consistency. For instance, the most basic constraint SVAI1 assures that each component modeled in the variability model also exists in the code base of the product line. This constraint prevents, for example, that components that aren't available anymore or are outdated and therefore have been removed from the file system are not forgotten to be purged in the variability model as well. The two constraints SVAI2 and SVAI3 cover the relations between components in the model, and the relations between Spring XML files in the file system (these in fact depend on relations between the Java Beans described in that Spring XML files). Both con-

straints assure that there aren't any unnecessary relations between components respectively and that no relations are missing in the variability model.

TABLE I. EXAMPLES OF GENERIC (G) AND SIEMENS VAI-SPECIFIC (SVAI) CONSTRAINTS

	Constraints	
	Name	Description
G1	List decision	A list decision must have at least two options to choose from
G2	Mandatory attribute	Mandatory attributes must not be empty
G3	Decision effect cycle	There must be no cycles caused by logical decision dependencies
G4	Visibility condition cycle	There must be no cycles caused by hierarchical decision dependencies (visibility conditions)
G5	Visibility condition self reference	A visibility condition must not contain the decision itself
SVAI1	Component matching	Each component in the variability model must exist in the product line code base
SVAI2	Component relation	Relations between components in the variability model must also exist in the product line code base
SVAI3	Java Bean relation	A relation between Java Beans must be represented in the variability model as a component relation
SVAI4	Variant type relation	Variant types must not have requires relations
SVAI5	Variant type occurrence	If two or more components are identical, all of them must contribute to a variant type component
SVAI6	Variant type consistency	Only identical components must contribute to a single variant type component

To illustrate how the defined constraints work we discuss constraint SVAI2 in detail by showing its high-level operation sequence: SVAI2 checks the necessity of *requires* relations between components. As illustrated in Figure 2, a *requires* relation between two components in the model is only needed if it is based on an existing dependency in the product line code base. Each component is "implemented" through a Spring file which in turn contains one or more Java Beans. If at least one Java Bean defined in Spring file 1 requires a Java Bean defined in Spring file 2, the relation on component-level is needed. Otherwise the consistency check will reveal the unneeded relation between component 1 and component 2.

This consistency constraint is not inherently complex to understand – indeed, most are of similar complexity. However, it is important to note that such consistency constraints may have to be evaluated many times in a model. For example, constraint SVAI2 needs to be evaluated for each *requires* relationship among two components and there are thousands of such *requires* relationships in our models.

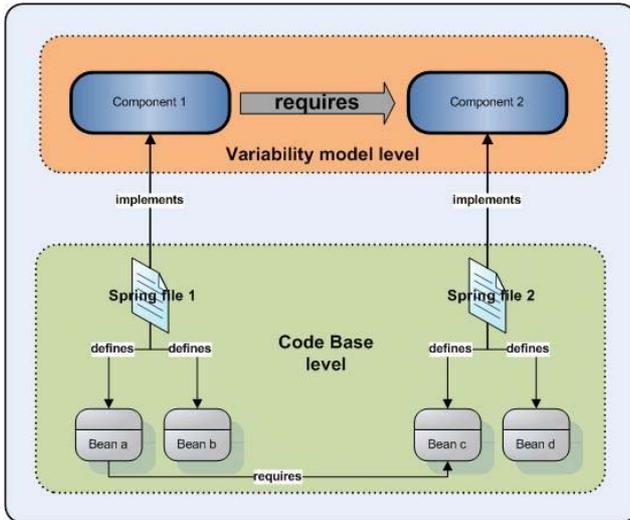


Figure 2. Schematic view of constraint SVAI 2

III. TOWARDS INCREMENTAL CONSISTENCY CHECKING SUPPORT FOR VARIABILITY MODELS

In our project with Siemens VAI, we developed a batch-oriented consistency checker early in the project. It worked fine as long as we were working with small variability models and a small number of constraints. The approach however didn't scale for very large models and a high number of required consistency checks. The performance problems did not allow to report inconsistencies to the user after each change to a model.

We therefore started exploring the use of an incremental consistency checker, which had been successfully evaluated for large UML models as part of the UML/Analyzer tool for instant consistency checking of UML models [8]. The tool helps designers in detecting and tracking inconsistencies correctly and quickly with every design change.

A consistency constraint needs to be re-evaluated if and only if one of the affected model elements changes. We refer to this set of model elements as the scope of a consistency constraint. Identifying the scope is simple in principle, however, it is not possible to predict in advance what model elements are accessed by any given consistency constraint.

The UML/Analyzer tool circumvents this problem by observing the run-time behavior of consistency constraints during their evaluation. To this end, the equivalent of a profiler for consistency checking was developed. The profiling data is used to establish a correlation between model elements and consistency constraints (and inconsistencies). Based on this correlation, it then decides when to re-evaluate consistency constraints and when to display inconsistencies – allowing an engineer to quickly identify all inconsistencies that pertain to any part of the model of interest at any time.

IV. INTEGRATING THE INCREMENTAL CONSISTENCY CHECKER IN THE DOPLER TOOL SUITE

We have been integrating the incremental consistency checker approach in the Eclipse-based DOPLER tool suite.

Figure 3 shows a high-level architecture of our tool and the main components of the checker.

The *DOPLER variability model editor DecisionKing* (#1) supports creating and updating variability models.

The *incremental consistency checker* (#2) performs constraint initialization, management, and persistence and applies incremental checking to variability models independent from the domain-specific meta-model used. This guarantees that the approach can later be easily used with other meta-models and is not limited to Siemens VAI.

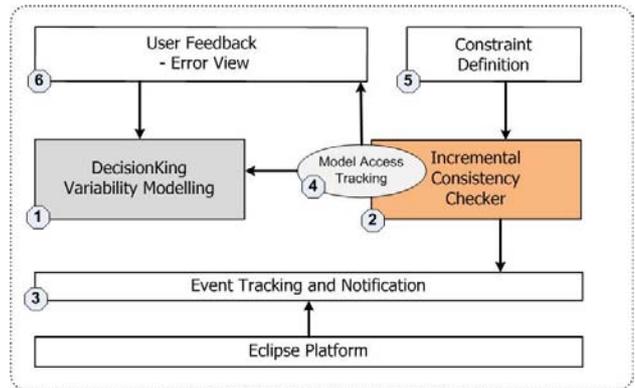


Figure 3. High-level tool architecture of incremental consistency checking within the DOPLER tool suite.

For instant consistency checking it is necessary to track user changes during modelling. An *event tracking and notification mechanism* (#3) allows observing changes to the variability model and the Eclipse workspace at a very detailed level. It provides information about DOPLER variability models being opened for editing and manages the propagation of change notifications from model elements to the incremental consistency checker [1]. As described in the previous section, incremental consistency checking highly depends on the ability of tracking and processing changes from various sources. The more fine-grained these change events can be tracked, the better performance can be achieved, because with each level of information detail fewer constraint instances eventually need to be evaluated. Our event tracking mechanism in the DOPLER tools allows to identify changes down to the level of model element attributes. Therefore, very few constraints need to be evaluated during incremental consistency checking.

The *model access tracker* (#4) monitors and logs all read access events to model elements for each single constraint instance. Each call on a model element by a constraint has to be done through a “model profiler”, which is capable of capturing and tracking any read access on attribute level. This evaluation profiling ensures that all necessary constraints are re-evaluated when a change event occurs.

The *consistency constraint definition* (#5) uses the Eclipse extension point mechanism to add constraints to the incremental checking tool as different application domains need specific constraints. Note that our approach distinguishes the definition of a constraint from its evaluation.

The *error view* (#6) provides feedback to the users on the inconsistencies detected for the evaluated constraint. Eclipse provides the marker mechanism, which allows for easy creation and management of occurring errors. To assure a high level of flexibility, evaluation results are also provided through the extension point mechanism to make them utilizable in other plug-ins or in custom views.

V. APPLICATION EXAMPLE

A major goal when developing our incremental consistency checker was to achieve a better usability and responsiveness for the modeler working with the DOPLER tool suite and detecting and providing information on inconsistencies as early as possible. In the following we will describe a brief scenario of how a modeler can work with the tool, and in which way the tool supports detecting and fixing inconsistencies.

Figure 4 shows a screenshot of the DecisionKing variability model editor: In the Asset Overview (#1), the modeler can find an outline of already defined assets. Assets can also be added or removed here. The Detail View provides information about the currently selected asset, their attributes (#2), as well as relations to other assets (#3). An error view (#4) provides information on errors in the variability model, i.e., inconsistencies.

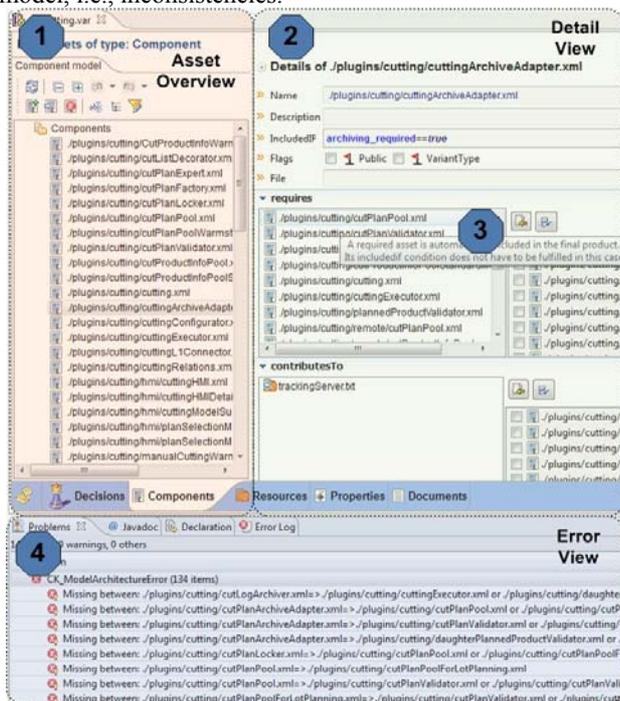


Figure 4. DecisionKing Variability Modeling Tool and Error View displaying inconsistencies found after changes to the model.

A typical modeling process starts with defining new assets that are relevant in the newly created model. After defining all needed assets, the relations among them need to be modeled. The modeler can add or remove relations to other assets via drag & drop in the detail view (#3). In contrast to the old batch-oriented approach, manipulating elements in

the model now has an immediate effect. After adding a relation to an asset all involved constraints are being re-evaluated. Feedback about an inconsistency in the model is provided in that second the user takes the wrong action. The error view (#4) then provides detailed information on the occurring inconsistency and the source responsible for that. Assisted by this, the modeler can draw conclusions and resolve the occurring inconsistency by (in this example) removing an unneeded relation from the model.

VI. CONCLUSIONS

We presented initial tool support for applying incremental consistency checking on variability models. Our experiences with large-scale models demonstrate the performance and scalability of the approach. In future work we will increase the number of types of constraints and will also investigate how the dependencies among constraints can be exploited to further improve performance. Moreover, we will analyze the performance of the approach in detail.

ACKNOWLEDGMENTS

This work has been conducted in cooperation with Siemens VAI Metals Technologies and has been supported by the Christian Doppler Forschungsgesellschaft, Austria. This work has also been supported by FWF grant P21321-N15

REFERENCES

- [1] K. C. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," TR CMU/SEI-90TR-21, Carnegie Mellon Univ., Pittsburgh, PA, USA 1990.
- [2] G. H. Campbell, Jr., S. R. Faulk, and D. M. Weiss, "Introduction To Synthesis," INTRO_SYNTHESIS_PROCESS-90019-N, Software Productivity Consortium, Herndon, VA, USA 1990.
- [3] F. Bachmann, M. Goedicke, J. C. S. do Prado Leite, R. L. Nord, K. Pohl, B. Ramesh, A. Vilbig: "A Meta-model for Representing Variability in Product Family Development". PFE 2003: 66-80.
- [4] B. Belkhouche and C. Lemus, "Multiple View Analysis and Design," Proc. of the Viewpoint 96: Int'l WS on *Multiple Perspectives in Software Development*, 1996.
- [5] B. H. C. Cheng, E. Y. Wang, and R. H. Bourdeau, "A Graphical Environment for Formally Developing Object-Oriented Software," Proc. of the 6th Int'l Conf. on *Tools with Artificial Intelligence*, New Orleans, USA, 1994, pp. 26-32.
- [6] A. Tsiolakis and H. Ehrig, "Consistency Analysis of UML Class and Sequence Diagrams Using Attributed Graph Grammars," Proc. of the *Graph Transformation & Graph Grammars*, Berlin, 2000, pp. 77-86.
- [7] D. Dhungana, R. Rabiser, P. Grünbacher, and T. Neumayer, "Integrated tool support for sw. product line engineering," Proc. 22nd *IEEE/ACM Int'l Conf. on Automated Sw Eng (ASE'07)*, pp. 533-534.
- [8] A. Egyed, "Instant Consistency Checking for the UML," Proc. of the 28th Int'l Conf. on *Sw. Eng. (ICSE)*, Shanghai, China, May 2006.
- [9] D. Dhungana, P. Grünbacher, and R. Rabiser, "Domain-specific Adaptations of Product Line Variability Modeling," Proc. of the *IFIP WG 8.1 Working Conf. on Situational Method Engineering*, Geneva, Springer Series in CS, 2007, pp. 238-251.
- [10] R. Johnson, J. Höller, and A. Arendsen, "Professional Java Development with the Spring Framework," Wiley Publishing, 2005.
- [11] W. Heider, R. Rabiser, D. Dhungana, P. Grünbacher, Tracking Evolution in Model-based Product Lines. 1st Int'l WS on *Model-driven Approaches in Software Product Line Engineering (MAPLE 2009)*, Proc. (vol 2) of the 13th Int'l SW. Product Line Conf. (SPLC 2009), San Francisco, CA, 2009, Carnegie Mellon Univ., pp. 59-63.

A Support Tool for Domain Analysis

Liana Barachisio Lisboa
RiSE - Reuse In Software Engineering
 Bahia - Brazil
 Email: liana@rise.com.br

Vinicius Cardoso Garcia
 Silvio Romero de Lemos Meira
RiSE and Federal University of Pernambuco
 Email: {vcg,srlm}@cin.ufpe.br

Eduardo Santana de Almeida
RiSE and Federal University of Bahia
 Bahia - Brazil
 Email: esa@dcc.ufba.br

Abstract—Nowadays, companies need to improve their competitiveness. Thus, they seek systematic ways of adopting software reuse, and domain analysis is one possibility to reach it. However, it involves the management and analysis of a large set of interrelated information from several systems. Hence, due to its complexity, a support tool is necessary. This paper, presents a tool called ToolDay, which aims at making the process semi-automatic and at aiding the domain analyst to achieve systematic reuse in an effective way. In addition, its evaluations are also described.

Keywords-Domain Analysis Tool, ToolDay, Evaluation

I. INTRODUCTION

Nowadays, companies are seeking for ways to improve their competitiveness, which involves less time-to-market and high quality for products. The adoption of software reuse is an option to obtain these benefits. Although the benefits of software reuse are promising, it is a complex task to put it into practice. A way to maximize these possible benefits is through a systematic reuse approach, which is domain focused, based on a repeatable process, and concerned with reuse of higher level life cycle artifacts [1]. One of the ways to accomplish this is through a domain analysis (DA) process, which is the process of identifying common and variable characteristics of systems in a specific domain.

The DA process is composed of some interdependent activities that involve the management of complex and interrelated information from various sources. Due to this, the use of human expertise in industrial projects without automation can contribute to risks in a project.

The development of ToolDay was based on a systematic review of DA tools that analyzed how existing tools supported the DA process [2]. In this review, the authors analyzed nineteen relevant tools to extract the results.

From the results, it was identified that tools usually come from the necessity of supporting a specific process instead of a generic one. However, this may force companies to modify or adapt established development processes, which can lead to a higher learning curve and a bigger impact on the company development life cycle [2].

Another outcome was the identification of a set of functionalities that any tool should have. They were extracted from the selected tools and grouped into phases, which were: (i) *Planning*, analyze systems to see what is valid or not

Table I
 FUNCTIONALITIES GROUPED BY PHASE WITH THEIR PRIORITIES

Functionality	Priority
Planning phase	
Pre Analysis Documentation	Low
Domain Matrix	Essential
Evaluation Functions	Low
Scope Definition	Important
Modeling phase	
Domain Representation	Essential
Variability	Essential
Mandatory Features	Essential
Composition Rules	Essential
Feature Group Identification	Low
Relationship Types	Low
Feature Attributes	Low
Validation phase	
Domain Documentation	Essential
Feature Documentation	Important
Requirements Management	Important
Relationship between Features and Requirements	Low
Dictionary	Important
Reports	Important
Consistency Check	Essential
Product Derivation	
Product Derivation	Important
Product Documentation	Important

to be included in the domain scope; (ii) *Modeling*, model the defined domain in a visual way; and (iii) *Validation*, document and validate the generated artifacts.

Furthermore, functionalities for the product derivation were also identified in the majority of tools. Finally, a total of twenty functionalities were recognized, which are shown in Table I with their priorities. With these analyses, the reviewers identified that there is not a tool that focus on all phases, and the majority of them offer support, mainly, to the modeling phase.

This tool development also came from industrial needs that were experienced along five years of projects involving software reuse at C.E.S.A.R¹, a Brazilian Innovation Institute with CMMI level 3.

This paper presents ToolDay, which goal is to support the DA process, making it semi-automatic and aiding the analyst to achieve systematic reuse in an effective way. This work

¹Recife Center for Advanced Studies and Systems - <http://www.cesar.org.br>

extends a previous one [3], in which the main functionalities of the tool (Table I) were described. This paper makes two contributions, (i) describing the second iteration of ToolDAY and (ii) reporting two evaluations.

II. TOOLDAY

Due to lack of space, several artifacts are just mentioned herein, to see the artifacts and some screen shots go to <http://www.cin.ufpe.br/~vcg/toolday>.

The DA process starts with the **planning phase**. ToolDAY provides a set of documentation fields for pre-analysis documentation that aids in the identification of what characteristics should be in the domain. The documentation includes: identifying the stakeholders; objectives and constraint definition; market analysis and data collection.

Then, the domain scope can be defined through the product map (domain matrix in Table I) that relates and compares characteristics of domain applications, extracted from pre-analysis documentation, to identify which ones should be part of the domain through some metrics, called evaluation functions. Their results, which can be *mandatory*, *variable* or *out of the scope*, influence the domain scope.

In the **modeling phase**, ToolDAY performs the domain representation with a feature model, in which the features are diagrams and their types are relationships (which can be alternative, or, optional and mandatory, plus the composition rules that can be implication and exclusion).

Also regarding the relationships between features, they can be represented in the model with different line formats. The types are *composition*, used if there is a whole-part relationship; *generalization*, when features are generalization of sub-features; and *implementation*, when a feature implements the other feature. There is also the default relationship that has no type.

Besides, features can be grouped according to the information they represent. The groups are identified through different colors in the feature border. They can be: *capability*, characterizes a distinct service or functionality a product may have; *operating environment*, represents an environment attribute in which the product is used; *domain technology*, corresponds to a domain specific implementation; and *implementation technique*, implementation details that can be reused cross domains.

ToolDAY also implements the inclusion of attributes for the features. They synthesize the representation of a large number of possible variations improving the understandability of the feature model.

For the **validation phase**, ToolDAY provides a large set of documentation. Apart from the domain and features documentation [3], ToolDAY permits the description of requirements and use cases, which are optional artifacts in project execution.

The *requirement* documentation includes priority, type (functional or non-functional) and description, while the *use*

case includes pre and post conditions and the execution flows: main, alternative and exception.

After specifying the requirements, use cases and features, it is possible to map the traceability among them. This facilitates the identification of the impact of one modification. This traceability is done in a visual model with different diagrams shapes for each artifact.

The consistency checker verifies if the relationships between features are correct. ToolDAY's consistency rules are divided in three categories: *redundancy*, the same semantic information is represented in more than one way. *Anomalies*, some features configurations are lost and the domain cannot be completely configurable. And, *inconsistency*, some representation contradicts with other information in the model. Each category has a set of verifications [4].

Moreover, ToolDAY also supports the inclusion of a dictionary, which purpose is to clarify the terms of the domain. Furthermore, there is no advantage in proving a large set of documentation, if they were only available within the tool environment. Thus, to provide their visualization in other environments, ToolDAY permits the creation of several reports in different formats (PDF, Excel or Images).

The **product derivation** with ToolDAY is done through the selection of the domain features. This selection occurs in a tree view representing the domain hierarchy. There is also a consistency checker for the product, but it has different validation rules [4] and all of them are classified as inconsistencies. Once the product is valid, the product model can be generated. In this model all features are mandatory and the user can include new features, usually the ones marked as "out of scope" in the product map. Each product has a simple documentation that includes the domain version it was based on and its description.

III. EVALUATION

To evaluate the described tool, two case studies were performed. The first was in a controlled environment, while the second was in a software company.

A. Controlled Environment

The case study followed guidelines from [5]. This study was performed before the second development iteration.

The goal of the study was to analyze ToolDAY with respect to its aid in the DA execution and easy to use environment. To achieve it, some questions were defined for the subjects. They were, (Q1) if ToolDAY aids in the execution; (Q2) if there were any difficulties using the tool; (Q3) if the consistency checker is helpful; and (Q4) if ToolDAY tutorial is enough to learn how to use it.

The study was conducted in a post-graduation course at a university lab from November, 2007 to February, 2008 by the students that performed a domain engineering (DE) project based on a real-world case.

The project consisted of four reuse tools, which provided solutions to increase the organization productivity through reuse according to its maturity level. At the end of the DA, they generated a feature model with 64 features, 14 requirements and 38 use cases.

The subjects were six students that played the domain analyst role. All of them had worked before with the same platform ToolDay uses (Eclipse²) and half of them knew some DA processes, while the other half knew just one.

After the project was concluded, the subjects were asked to fulfill a feedback form with questions related to it. Their answers are described next.

Q1: Four subjects considered that the tool aided in the process execution, another judged that the tool did not help a lot during the process execution, and the other did not see the gain in using the tool.

The reasons given by the two subjects for not considering the tool helpful were: great part of the process can be done without tool support; lack of integration with the next steps of DE; and the tool is not completely integrated with the DA process used. However, ToolDay focuses on the DA process support, a few steps of the product derivation, and on generating the artifacts that will be later used on the DE phases, it does not intend to integrate the complete process. The other subjects explained why they considered the tool helpful: it helps the process execution steps; and it aids in the scope definition and in the domain modeling.

They also described some weaknesses and strengths about the tool. The weaknesses were: traceability among requirements, use cases and features is too simple (in the evaluation, requirements and use cases only had the name and description and the traceability editor did not exist); lack of requirement management; and a model with too many features becomes too polluted. Some of the strengths were the consistency checker; generation of reports; and the visual representation of the domain model.

Q2: Only one subject did not have difficulty with the tool. The other answers were (multiple answers per subject): Two said that the navigation is hard (both are familiar with the platform and some DA processes). Two answered that there was lack or insufficient explanation for using the tool (both are familiar with the platform, but one knows few DA processes and the other just one). A subject related the lack of knowledge in the DA process (he knows only one DA process). In addition, a few subjects also informed that the difficulty occurred because of the symbols used to represent the relationships and the traceability between the domain feature model and the product map.

Even though several subjects reported at least a difficulty, they were mostly related to GUI or to specific aspects of traceability (which have already been improved) and not to the main functionalities of the tool.

²<http://www.eclipse.org>

Q3: Only one subject informed that the consistency checker did not fully aid the problem identification. The restriction was the lack of an easier identification of where the problem is, i.e. the tool should select the exact spot of the problem. The others considered it sufficient for identifying and resolving the problems.

Q4: Four subjects considered the information of the tutorial sufficient for learning how to use the tool and the other two subjects did not use it. In addition, one that did not use it was the same who declared that had some difficulty due to lack or insufficient explanation for using the tool. Therefore, it indicates that the tutorial may overcome this.

Even with the analysis not being conclusive, the study indicated that the tool has some strength for the DA support. On the other hand, aspects related to understanding (difficulties during the process execution) were the focus of the second iteration.

Nevertheless, some of the problems described by the subjects can be resolved if a proper training, before the tool starts to be used, is performed. After concluding the study, some aspects should be considered before repeating it.

Training. Instead of the subjects learning how to use it through the tutorial, a basic training can be applied. The training can emphasize on unused aspects by the subjects of this study and on the complaints related to it.

Questionnaires. The questionnaires should be reviewed to collect more precise data related to where the problems and difficulties occurred.

According to the feedback, some requirements were identified and developed in the second iteration. The consistency checker now selects the exact spot where the problem is. The requirements and use cases (as described before) are more detailed and permit the traceability with features. The traceability model can be exported as an excel document. Additionally, some visualization filters to models with too many features were created.

The other improvements from the questionnaires, such as import the features added in the domain feature model to the product map and search for features have not been implemented yet.

B. Industrial Case Study

The industrial case was developed at C.E.S.A.R. and it is part of the company software reuse effort, as a way to institutionalize reuse in all of its projects.

The business goals defined for the project were: (a) increase the productivity and (b) reduce maintenance costs and development efforts. The pilot project selected was a *web/social network* with seven different releases. The project goal was to adopt a software product line with the benefits of: (i) better understandability of the project business; (ii) identification of new market opportunities; (iii) identification of new functionalities for the product; and (iv) decrease the maintenance cost.

The team goal for the DA phase was to identify existing features from other tools that were not present in their tool. They performed the DA documenting the domain, defining its scope and building a glossary. They created the product map of the analyzed applications and the features model of domain with 74 features.

At the end, the team considered that the DA goal was achieved. Moreover, a better understanding of the domain being developed was accomplished, since several new features were identified and planned to be developed.

The domain analyst did not have any formal DA process, therefore he followed ToolDAY steps and had no difficulty in using it. However, improvements were highlighted, such as the possibility to export the product map as a table.

Even though the goal was achieved and the process result brought benefits to the team, for the domain analysts there was no real data that using ToolDAY during the DA process aided it. However, it is necessary to highlight that since the user did not follow a specific process, the tool contributed in the process execution because it provided a guideline to define the domain scope, its feature model and to document them. Furthermore, this industrial case worked as a proof of concept for the tool.

IV. RELATED WORK

The systematic review [2] identified nineteen tools supporting the DA processes. Some concentrate on the planning phase - like PuLSE-BEAT [6] and DREAM [7] - while others on the modeling - such as RequiLine [8]. The support for the validation phase differentiates among them, from almost all to none support. Two tools outstand in the review according to the number of essential requirements they support, Holmes and RequiLine.

Holmes [9] provides functionalities for all identified phases. It permits pre-analysis, domain documentation, composition rules support and validation, along with the domain scope and a visual representation of the domain. Even though it supports product instantiation, no documentation is provided to it and to the features.

RequiLine [8] supports almost all functionalities for the modeling phase, except feature group identification. It provides a vast documentation for the domain and features, and permits the inclusion of requirements that can be related with the features in the domain. Besides, it supports product derivation and documentation.

Among the analyzed tools, none of them provides a full support to the process. The two closer to it still lack few functionalities. ToolDAY development was planned focusing on this problem and its development, according to results of the review, is fulfilled.

V. CONCLUSION

This paper presented ToolDAY, a tool that offers support to several requirements within the process, which includes

scope definition, domain modeling, documentation and consistency. Furthermore, ToolDAY also supports requirements for the product derivation.

The support to requirements in every phase and having the whole support in the same environment is one of ToolDAY advantages when compared to other DA tools. Even though it has a defined process, it can be used without following it, since the artifacts of planning and modeling (domain and product) can be independently built.

The evaluations highlighted improvements that are being developed. Furthermore, they indicate that the tool aids in the process support, but new studies are necessary. For future work, ToolDAY is being extended with feature interactions; metrics regarding the domain information and different visualization views for the domain representation.

ACKNOWLEDGMENT

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES³), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08.

REFERENCES

- [1] W. B. Frakes and S. Isoda, "Success factors of systematic reuse," *IEEE Software*, vol. 11, no. 5, pp. 14–19, 1994.
- [2] L. B. Lisboa, D. Lucrdio, V. C. Garcia, E. S. Almeida, S. R. L. Meira, and R. P. M. Fortes, "A systematic review of domain analysis tools," *Journal of Information and Software Technology*, vol. 52, pp. 1–13, 2010.
- [3] L. B. Lisboa, V. C. Garcia, E. S. Almeida, and S. L. Meira, "Toolday - a process-centered domain analysis tool," in *Brazilian Symposium on Software Engineering - Tools Session*, Brazil, 2007, pp. 54–60.
- [4] T. v. d. Massen and H. Lichter, "Deficiencies in feature models," in *Workshop on Software Variability Management for Product Derivation*, EUA, 2004.
- [5] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.
- [6] K. Schmid and M. Schank, "Pulse-beat – a decision support tool for scoping product lines," in *Software Architectures for Product Families*, Spain, 2000, pp. 65–75.
- [7] M. Moon, K. Yeom, and H. S. Chae, "An approach to developing domain requirements as a core asset based on commonality and variability analysis in a product line," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 551–569, 2005.
- [8] T. v. d. Massen and H. Lichter, "Requiline: A requirements engineering tool for software product lines," in *International Workshop on Product Family Engineering*. Italy: Springer Verlag, 2003, pp. 168–180.
- [9] G. Succi, J. Yip, E. Liu, and W. Pedrycz, "Holmes: a system to support software product lines," in *International Conference on Software Engineering*. Ireland: ACM, 2000, p. 786.

³INES - <http://www.ines.org.br>

Research Tool to Support Feature Configuration in Software Product Lines

Ciarán Cawley, Patrick Healy, Goetz Botterweck

Lero
University of Limerick
Limerick, Ireland

{ ciaran.cawley | patrick.healy | goetz.botterweck } @lero.ie

Steffen Thiel

Department of Computer Science
Furtwangen University of Applied Sciences
Furtwangen, Germany
steffen.thiel@hs-furtwangen.de

Abstract— Configuring a large Software Product Line can be a complex and cognitively challenging task. The numerous relationships that can exist between different system elements such as features and their implementing artefacts can make the process time consuming and error prone. Appropriate tool support is key to the efficiency of the process and quality of the final product. We present our research prototype tool which takes a considered approach to feature configuration using visualisation techniques and aspects of cognitive theory. We demonstrate how it uses these to support fundamental feature configuration tasks.

Keywords—visualisation; variability management; software product lines;

I. INTRODUCTION

Configuring a Software Product Line (SPL) with thousands of variation points in order to derive a specific product variant is a challenging process. Each configurable feature can have numerous relationships with many other elements within the system. These relationships can impact greatly on the overall configuration process. Understanding the nature and impact of these relationships during configuration is key to the quality and efficiency of the configuration process [1].

Information Visualisation techniques have provided a variety of ways for stakeholders to view, comprehend and manage large amounts of related information [2, 3]. However, although recent work has attempted to incorporate these into the domain of variability management [4-6], there appears to be a lack of their explicit consideration in current tools.

In this paper, we present a research prototype tool, which combines aspects of cognitive theory with specific visualisation techniques to provide alternative interactive views on the underlying data.

II. TOOL

The tool has been implemented as an Eclipse Plugin [9] providing a set of synchronised views that allow the loading, exploration, comprehension and manipulation of the underlying data models. These interactive views are designed with the aim of providing cognitive support to the stakeholder during feature configuration. Three distinct approaches have been employed - 2D, 2.5D and 3D.

A. Meta-Model

A data meta-model is used as the basis for our visualisation approach. It consists of three separate but integrated meta-models and describes a product line in terms of *Decisions*, *Features* and *Components*:

- A *decision* model captures a small number of high-level questions and provides an abstract, simplifying view onto *features*.
- A *feature* model describes available configuration options in terms of “prominent or distinctive user visible aspects, qualities, or characteristics” [11].
- A *component* model describes the implementation of *features* by software or hardware components.

These three models are interrelated. For instance, making a *decision* might cause several implementing *features* to become selected, which in turn require a number of *components* to be implemented. The meta-model also defines intra-model relationships such as *feature requires feature* or *feature excludes feature*. The details of this meta-model are out of scope for this paper and the interested reader is guided to a previous publication [10] for further information.

B. Task Support

As the end result of this work is to provide support to stakeholders during the feature configuration stages of SPL product derivation, we set out the tasks for which this support is being provided.

The activity of configuring a *feature* is the fundamental task challenging a stakeholder during the feature configuration process. At a basic level, this involves the ability to either include or exclude a *feature* from the product under derivation. We would also add that the ability to include/exclude *features* in groups based on higher level requirements (*decisions*) is also a fundamental task. Whereas these tasks may seem simplistic, it is the knowledge/understanding (cognition) of the stakeholder that allows these tasks to be performed correctly. Drawing on work carried out by others [1, 12], we outline a set of simple cognitive tasks that aim to support the activity of the primary task – to decide which *features* should be included and which should be excluded.

1. Identify / Locate a configuration *decision*
2. Understand the high-level impact of a *decision* inclusion (perception of scale and nature of the impact - implements/requires/excludes)
3. Identify / Locate a specific *feature*
4. Identify a specific *feature's* context - parent *feature*, alternative/supporting *features*, *sub-features*
5. Understand the high-level impact of a *feature* inclusion - a specific *feature's* constraints (requires/excludes relationships)
6. Identify the state of a *feature* - included/excluded and why.

It is these cognitive tasks that our visualisation approaches target in terms of providing an interactive visual environment.

C. Interactive Views

1) *2D Approach*: Using 2D approaches such as matrices and graphs to visualise feature models is the traditional way to allow feature exploration and model manipulation [5, 10]. In our 2D approach we provide a linear horizontal tree as the basis upon which we apply a number of visualisation techniques to support the configuration process. The tree view was implemented using the prefuse visualisation toolkit [13].

Figure 1 presents a screenshot from our Eclipse [9] based tool showing our 2D visualisation. For this 2D approach (and also for the subsequent 2.5D and 3D approaches), a supporting

synchronised view is used. This view in the left of the figure presents a simple list view of the *decisions* that identify the high level functionality/requirements that the system implements.

Through selection of a *decision* in the supporting view by mouse-click, the main tree view in the centre of the figure displays all implementing *features*, their location within the *feature* model and their immediate *sub-features*. *Animation* is employed during the tree view transition from its previous visual state to preserve the context. The tree itself is a *Degree of Interest* tree and automatically displays *features* of interest (path to current node, sibling nodes and child nodes) to the current selection and hides all other *features*. The combination of multiple windows and *Degree of Interest* aim to provide *Focus+Context*.

Colour encoding is employed to highlight what *features* directly implement (amber) the selected *decision* and what *features* are required (blue) or excluded (red) by those implementing *features*. A colour encoded icon (sphere) to the left of the label of a highlighted *feature* identifies if the *feature* has been included (green), eliminated (grey) or is un-configured (yellow).

The stakeholder can explore the tree through mouse-clicks on nodes of interest. Again the tree, using smooth animation, automatically expands and collapses nodes depending on the selected node of interest. The collapsing/hiding of nodes while exploring the tree can be stopped at the will of the stakeholder to allow manual collapsing and expanding of branches. Using

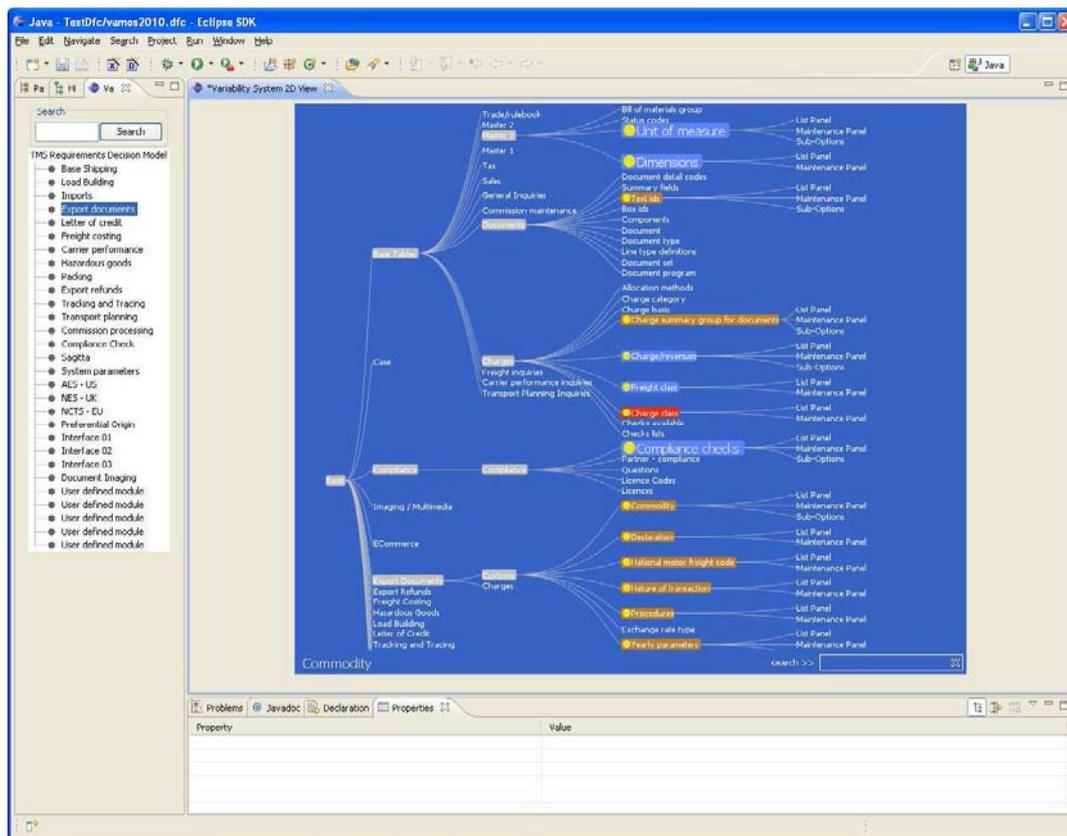


Figure 1 2D Tree View

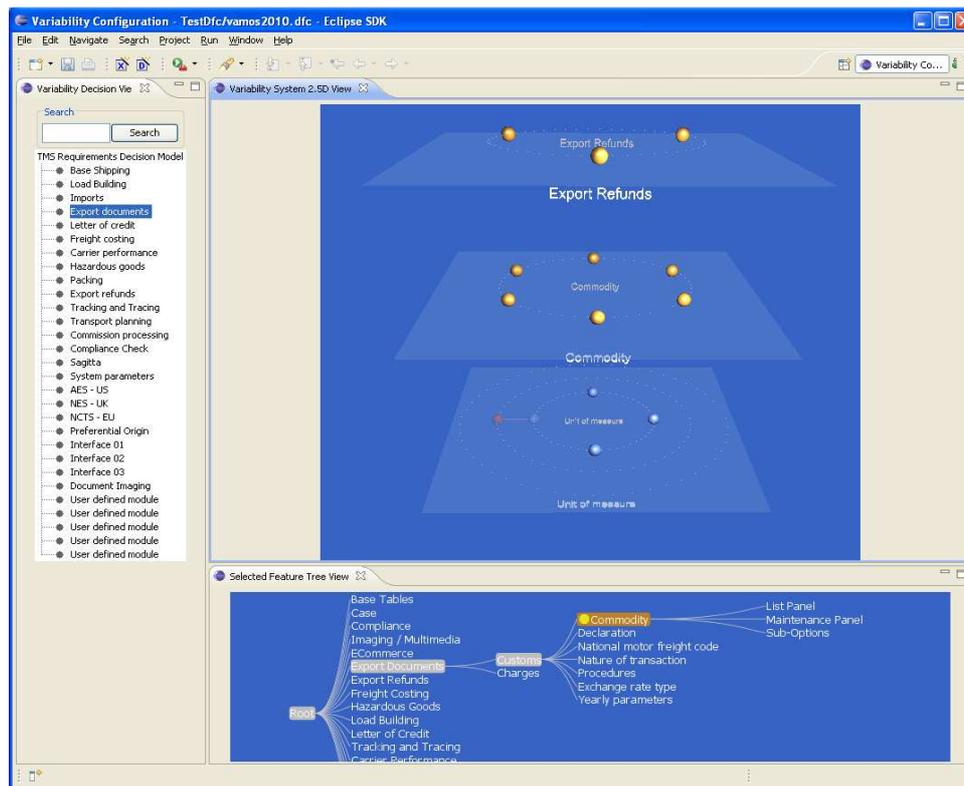


Figure 2 2D Planar View

the mouse, the stakeholder can perform full *zoom* and can also *pan* the entire tree in any direction. These functions aim to implement the *Details On Demand* principle.

2) *2.5D Approach*: 2.5D is a term that describes the use of 3D visual attributes in a 2D display [14]. For example, adding 3D attributes such as perspective (e.g. making certain objects smaller to indicate distance) and occlusion (e.g. overlapping objects to indicate layers) to a 2D display can be described as creating a 2.5D display.

Figure 2 presents our 2.5D view. Again, when a selection is made within the supporting *decision* view, the main view displays the implementing *features* along with all *features* that are required or excluded by them.

The view, inspired by Robertson et al.'s cone trees [15], consists of three stacked planes. Each plane provides a circular grouping of spheres. In the top plane, each sphere in the circle represents a grouping of *features*. When any one of those groupings in the top plane is selected (by mouse-click) then all *features* that comprise that grouping are displayed in the middle plane in a similar circular format. In the lower plane, all related (required / excluded) *features* are displayed (for *all features* presented in the middle plane). The innermost circle on the lower plane identifies *features* that are directly related (required, excluded) to *features* in the middle plane. In order of ascending radii, each subsequent circle in the lower plane represents the transitive relationships that exist i.e. required *features* can further require and/or exclude other *features*. In Figure 2 the stakeholder has selected the "Export Refunds" grouping in the top plane which groups six *features*. These six

features are represented on the middle plane while their related *features* (required, excluded) are represented on the lower plane.

By *hovering* the mouse over any sphere in any of planes, a description of that element will be displayed in the centre of the plane. When a sphere is *selected* in any plane, the circle on which it is presented will rotate so that that sphere is brought to the front with its description displayed underneath. These functions aim to implement *Details on Demand*.

The *colour encoded* sphere acts as the representation of a *feature* and its relationship. An amber sphere indicates a *feature* that implements the current *decision* selection. A blue sphere indicates a required *feature* while a red sphere indicates an excluded *feature*.

Multiple windows (and multiple planes) are employed to separate and distribute *decisions*, *feature* groupings, *features* and *relationships*.

3) *3D Approach*: Differing reports exist on the effectiveness of 3D visualisations to support software engineering but literature suggests that there is acceptance that it can be effective in specific instances.

Figure 3 presents a 3D view which attempts to provide a self contained representation of all three models (*decisions*, *features* and *components*) and their inter-relationships. However, at any given time, only information of interest is displayed.

Multiple windows (not shown) are employed to distribute the information and provide the supporting *decision* view.



Figure 3 3D Plot View

Figure 3 consists of a 3D space containing X, Y and Z axes. A sequential list of the *decisions* is displayed along the vertical Y-axis, a sequential list of the *features* along the horizontal X-axis and a sequential list of all the *components* along the Z-axis (moving away from the observer).

The key idea here is that a point within this 3D space identifies a relationship between all three models. In other words, a sphere plotted at a particular point will identify that the *feature* labelled at its X co-ordinate implements the *decision* labelled at its Y co-ordinate and is implemented by the *component* labelled at its Z co-ordinate. In Figure 3, the stakeholder has highlighted the sphere that represents the “Commodities” *feature*. However, in addition to this, by looking at the highlighted labels on the axes, we can see that it also represents the “Export Documents” *decision* that the *feature* implements and the “XTCM.I Include File” *component* that implements the *feature*.

Focus+Context and *Details On Demand* are the main techniques guiding this implementation. We argue that all three models can be perceived to be represented through the listings on each axis. However, the details of any part of any model or its relationships are only displayed when required. For example, when a *decision* is selected there can be a number of implementing *features*. For each implementing *feature*, a sphere is plotted in the 3D space as described above. Other *features* that are required or excluded by those implementing *features* are also similarly plotted as spheres and are given a specific *colour encoding* - required *features* are blue and excluded *features* are red.

Pan & Zoom are combined with *rotation* to allow a full *world-in-hand* manipulation of the view in three dimensions letting the stakeholder position the view depending on the information of interest.

III. CONCLUSION

In this paper we have presented a research tool prototype that employs aspects of cognitive theory and visualisation techniques to support some of the fundamental but challenging tasks that exist when configuring large software product lines.

ACKNOWLEDGMENT

This work is partially supported by Science Foundation Ireland under grant number 03/CE2/I303-1.

REFERENCES

- [1] S. Deelstra, M. Sinnema, and J. Bosch, "Product Derivation in Software Product Families: A Case Study," *Journal of Systems and Software*, vol. 74, pp. 173-194, 2005.
- [2] S. K. Card, J. D. Mackinlay, and B. Shneiderman, *Readings in Information Visualisation: Using Vision to Think*. Morgan Kaufmann, 1999.
- [3] C. Ware, *Information Visualisation: Perception for Design*, 2nd ed.: Morgan Kaufmann, 2004.
- [4] F. Heidenreich, I. Savga, and C. Wende, "On Controlled Visualisations in Software Product Line Engineering," in *2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPL 2008)* Limerick, Ireland, 2008.
- [5] R. Rabiser, D. Dhungana, and P. Grünbacher, "Tool Support for Product Derivation in Large-Scale Product Lines: A Wizard-based Approach," in *1st International Workshop on Visualisation in Software Product Line Engineering (ViSPL 2007)* Tokyo, Japan, 2007.
- [6] D. Sellier and M. Mannion, "Visualizing Product Line Requirement Selection Decisions," in *1st International Workshop on Visualisation in Software Product Line Engineering (ViSPL 2007)* Tokyo, Japan, 2007.
- [7] pure-systems GmbH, "Variant Management with pure::variants," <http://www.pure-systems.com>, Technical White Paper, 2003-2004.
- [8] Biglever Software, "Gears," <http://www.biglever.com>.
- [9] "Eclipse IDE," <http://www.eclipse.org>.
- [10] G. Botterweck, S. Thiel, D. Nestor, S. B. Abid, and C. Cawley, "Visual Tool Support for Configuring and Understanding Software Product Lines," in *The 12th International Software Product Line Conference (SPLC08)* Limerick, Ireland, 2008.
- [11] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21," Software Engineering Institute, Carnegie Mellon University 1990.
- [12] M. Sinnema, O. d. Graaf, and J. Bosch, "Tool Support for COVAMOF," in *Workshop on Software Variability Management for Product Derivation - Towards Tool Support*, 2004.
- [13] J. Heer, S. K. Card, and J. A. Landay, "prefuse: a toolkit for interactive information visualization," in *Conference on Human Factors in Computing Systems* Portland, Oregon, USA, 2005.
- [14] C. Ware, "Designing with a 2 1/2D Attitude," *Information Design Journal*, vol. 3, pp. 255-262., 2001.
- [15] G. G. Robertson, J. D. Mackinlay, and S. K. Card, "Cone Trees: animated 3D visualizations of hierarchical information," in *Conference on Human Factors in Computing Systems* New Orleans, Louisiana, United States: ACM New York, NY, USA, 1991.

SMARTFORM: A Web-based Feature Configuration Tool

Wonseok Chae
Toyota Technological Institute at Chicago
wchae@tti-c.org

Timothy L. Hinrichs
University of Chicago
tlh@uchicago.edu

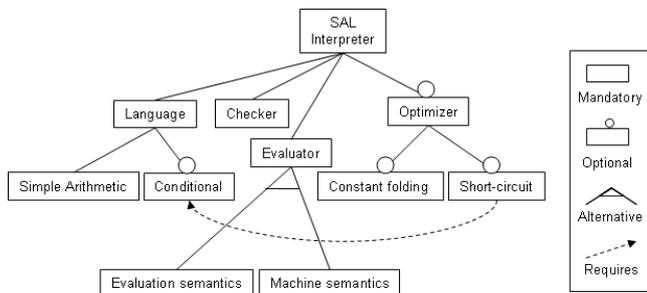


Fig. 1. Feature model for the SAL interpreter (from [4]).

Abstract—In feature-oriented software development, features distinguish different applications and a feature model abstractly represents the set of all possible applications for a given domain. Selecting a set of features that obey the feature model is the first step toward the synthesis of an application. In this paper, we present SMARTFORM, a web-based feature configuration tool that facilitates the process of feature selection. Its web-based feature selection user interface utilizes a web form generation tool PLATO to perform real-time validation.

Keywords—feature modeling, variability, configuration.

I. INTRODUCTION

Feature-oriented software development (FOSD) is an emerging paradigm for designing an entire domain of applications from a set of reusable software components. Each application in the domain is composed from a different subset of those components, and features are used to distinguish each application in the domain. Feature models represent all the permitted combinations of features, and selecting features that obey the feature model is the first step toward the synthesis of an application [10].

In previous work, we demonstrated that FOSD is an effective way to build a family of programs and showed that the process of combining software components given a feature selection can be automated using advanced implementation techniques [3], [4]. Those techniques assume that the given feature selection is valid, i.e., that it obeys the feature model, but when feature models are large or complex, this assumption can be difficult to satisfy.

One approach that helps significantly is making users understand the feature model before they begin feature selection.

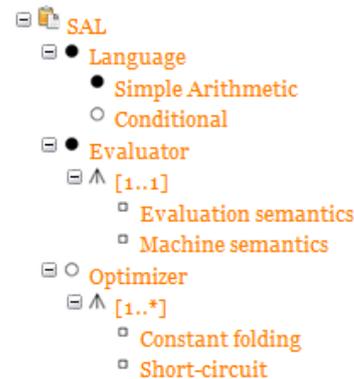


Fig. 2. SPLIT (Software Product Lines Online Tools) provides a feature modeling editor similar to that of FeaturePlugin [14]. Cross-tree constraints are separately specified in a conjunctive normal form. In this example, $\neg (\text{Short} - \text{circuit}) \vee \text{Conditional}$ specifies the constraint that the selection of Short - circuit requires the selection of Conditional.

The traditional representation of a feature model (a feature diagram, e.g. Figure 1), is well-recognized to be inadequate for large examples, though few alternative representations have been introduced. One notable exception is the FeaturePlugin tree representation, e.g., Figure 2, which allows for efficient entry and editing of large feature diagrams [1]. Unfortunately, for certain product domains, feature models can be so large that even the tree representation has been found inadequate. Additionally, even if the feature model is well-understood, it can be time-consuming to manually validate a given feature selection, especially when the feature model includes additional constraints (e.g., the selection of one feature precludes or requires the selection of others). In response some researchers have investigated techniques for reducing the complexity of feature models (e.g., this year's workshop on Scalable Modeling Techniques on Software Product Lines [13]), while others have focused on tools and methodologies that relieve both users and application developers from performing the task.

Automated tools have the potential to be of great benefit to users attempting to make valid feature selections. One branch of research is devoted to techniques for automated feature selection [15], though there is evidence that users would prefer to work interactively with feature-solicitation systems so they retain control over the feature selection process while leveraging the power of automation [9]. Another branch of

research aims at automatically validating a feature selection against a feature model. For example, recent work shows how off-the-shelf tools (e.g., the SVM systems, SAT solvers or CSP solvers) can be used to validate a feature selection by transforming the feature model to an appropriate decision problem [12], [2], [15], [8], [6]. This approach has the drawback that off-the-shelf tools are often difficult to deploy in settings for which they were not designed. For example, consider building a web-based collaborative feature selection tool where many users can simultaneously make (partial) feature selections and have those selections validated against a feature model automatically. To use an off-the-shelf SAT solver for feature validation, the SAT solver would need to be installed and run on the server. Thus under heavy usage, the server might be required to solve tens or even hundreds of SAT problems simultaneously, and since solving a single SAT problem is NP-hard, such an approach to collaborative feature selection would not scale.

In this paper, we explore the groundwork for building a web-based feature configuration tool that facilitates the process of feature selection. We render a feature digram in tabular form and employ checkboxes and radio buttons to allow multiple and singleton selections, respectively. This basic interface appeals to users not familiar with feature diagrams but tasked with making feature selections. Implementationally, our application offloads the work of feature selection validation to the clients, ensuring scalability. Moreover, feature selection validation is performed each time a user changes any portion of her selection, thereby allowing a user to understand at each choice point how that choice affects the overall validity of her selection. Our tool, SMARTFORM, builds upon a generic web form generation tool, PLATO [7], which constructs forms that provide users visual feedback about errors and implied values as they enter data.

II. PLATO: A FORM GENERATION SYSTEM

PLATO is a tool for automatically generating web forms from logical descriptions of those forms. A web form designer declares the desired properties for each form field and specifies the relationships between fields in logic. PLATO compiles the designer’s description into HTML and Javascript, complete with error-checking and value-propagation code. To understand the underlying mechanisms, it is instructive to first look at an example.

TABLE I
THE DECLARATION OF FORM FIELDS.

Name	Style	Type	Description
C	checkbox	boolean	Conditional
E	radio	enum {Evaluation, Machine}	Evaluator
O	checkbox	boolean	Optimizer
CF	checkbox	boolean	Constant folding
SC	checkbox	boolean	Short-circuit

Consider building a web form for the SAL interpreter feature model. First, we decide how to represent the features



Fig. 3. An example output of PLATO, which converts form descriptions into HTML and Javascript. An erroneous feature selection and its causes are displayed in different colors.

in the model as fields on the web form. Checkboxes are useful for optional features and radio buttons are good for alternative features. Table I includes five form fields (C, E, O, CF, SC) representing six features (Conditional, Evaluation, Machine, Optimizer, Constantfolding, Short – circuit).

Second, we specify all the constraints that the feature model imposes on the web form fields. In PLATO, constraints are written in a well-behaved fragment of first-order logic (FOL). Our running example requires three constraints:

- The selection of Optimizer requires the selection of either Constant folding or Short – circuit. At the same time, unless Optimizer is selected, neither of its children can be selected. These constraints are specified as follows:

$$(\Leftarrow O \text{ (or CF SC)})$$

Here, O, CF and SC refer to the name of fields we defined in Table I. *or* and \Leftarrow are logical operators, which give the logic above the following meaning. “O is true if and only if either CF or SC or both are true.”

- The selection of Short – circuit requires the selection of Conditional.

$$(\Rightarrow SC C)$$

- Either Evaluation semantics or Machine semantics must be selected. This constraint is enforced directly by the radio button but could be included explicitly.

PLATO compiles the two items above (field descriptions and logical constraints) into HTML and Javascript that implements error-detection and value-propagation code. Figure 3 shows the result that PLATO generates. It also illustrates an invalid feature selection (selecting Short – circuit without selecting Conditional and Optimizer) and the visual cues PLATO uses to identify errors and their causes.

III. SMARTFORM

The SMARTFORM tool is structured as shown in Figure 4. It takes a feature model written in a simple XML feature model format (SXF) and produces a web form for feature selection that performs validation each time the user makes a selection. Internally, SMARTFORM transforms the feature model into a logical web form description and passes the result to PLATO. It also generates layout information, which when combined with PLATO’s output produces the final web

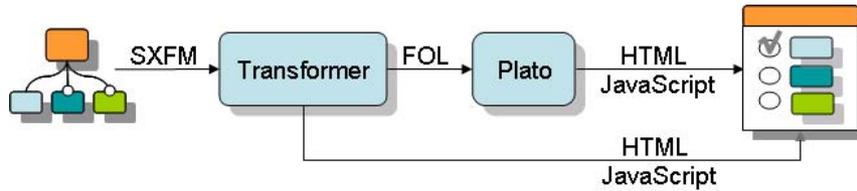


Fig. 4. The overall system for the SMARTFORM generator.

form for feature selection. In this section, we discuss each component of SMARTFORM.

A. Simple XML Feature Model

The Generative Software Development Lab at the University of Waterloo launched the Software Product Lines Online Tools (SPLIT) to put software product line research into practice [14]. They also introduced a simple XML feature model format (SXF) that makes it easy to define feature models using a simple text editor and illustrated their format with a library of about 30 models. So that we can utilize these models and their advanced feature model editor, we use the SXFM format as our primary form of input.

Figure 5 shows how our running example can be specified in SXFM. It consists of three sub-elements: meta, feature_tree and constraints. The meta section is self-explanatory. The feature_tree section corresponds to a feature diagram where the levels in the diagram are represented by tabulations instead of tags to keep the size of XML files small. The constraints section defines constraints not represented by the feature diagram directly in conjunctive normal form. In this example, the constraint that “the selection of Short – circuit requires the selection of Conditional” is represented by “`~_r_3_9_11 or _r_1_5`” simply meaning that “`_r_3_9_11` implies `_r_1_5`”.

B. Transformer engine

The Transformer engine parses a feature model in SXFM format into (i) HTML layout templates and (ii) a logical web form description for PLATO. The first item, conversion of the feature diagram to a tabular structure is straightforward. The only subtlety that arises is when the children of some parent are constrained as optional selections or as alternative selections. We treat such children as checkboxes and radio buttons, respectively.

As illustrated in Section II, PLATO requires a logical description of the form fields and the constraints on those fields. The web form description consists of three sections: widgets, types, and constraints. See Figure 6. A widget declaration defines the display properties for each form field. PLATO supports various styles (e.g., checkboxes and radio buttons), but since SMARTFORM generates its own layout information, this section can be ignored. The type section requires information about the possible values for each field: string, boolean, or an enumerated list, information that is easily extracted from the SXFM feature model. The constraints section is the most interesting section to construct. In addition to the “constraints”

```

<feature_model name="SAL">
<meta>
<data name="description">Simple Arithmetic Language</data>
</meta>
<feature_tree>
:r SAL(_r)
:m Language(_r_1)
:m Simple Arithmetic(_r_1_4)
:o Conditional(_r_1_5)
:m Evaluator(_r_2)
:g (_r_2_6) [1,1]
: Evaluation semantics(_r_2_6_7)
: Machine semantics(_r_2_6_8)
:o Optimizer(_r_3)
:g (_r_3_9) [1,*]
: Constant folding(_r_3_9_10)
: Short-circuit(_r_3_9_11)
</feature_tree>
<constraints>
constraint_2: ~_r_3_9_11 or _r_1_5
</constraints>
</feature_model>

```

Fig. 5. This model is created online using SPLIT’s Feature Model Editor.

element of the SXFM format, SMARTFORM includes structural constraints implicit in the feature model such as parent-child relationships. These structural constraints must be made explicit for PLATO since it is a general-purpose tool and includes no information about the semantics of feature-models.

In our example, Optimizer is the parent of Constant folding and Short – circuit, which causes SMARTFORM to include the constraint requiring at least one of Constant folding or Short – circuit to be selected if Optimizer is selected.

Once the logical web form description has been assembled, the transformer performs two tasks: it computes the HTML layout of radio buttons and checkboxes and sends the web form description to PLATO through a SOAP request.

C. SMARTFORM

SMARTFORM embodies the combination of SXFM, PLATO, and the Transformer engine. The results of the transformer engine and PLATO are combined to construct a web form for feature selection where feature validation is performed in real-time by the web browser. For instance, when a user selects Short – circuit without selecting Conditional and Optimizer, the form immediately highlights the violations, pointing to the

```
(WIDGET :NAME _r_1_5 :INIT "" :DESC "Conditional" :STYLE CHECKBOX ) (TYPE _r_1_5 BOOLEAN)
(WIDGET :NAME _r_2_6_7 :INIT "" :DESC "Evaluation semantics" :STYLE CHECKBOX ) (TYPE _r_2_6_7 BOOLEAN)
(WIDGET :NAME _r_2_6_8 :INIT "" :DESC "Machine semantics" :STYLE CHECKBOX ) (TYPE _r_2_6_8 BOOLEAN)
(WIDGET :NAME _r_3 :INIT "" :DESC "Optimizer" :STYLE CHECKBOX ) (TYPE _r_3 BOOLEAN)
(WIDGET :NAME _r_3_9_10 :INIT "" :DESC "Constant folding" :STYLE CHECKBOX ) (TYPE _r_3_9_10 BOOLEAN)
(WIDGET :NAME _r_3_9_11 :INIT "" :DESC "Short-circuit" :STYLE CHECKBOX ) (TYPE _r_3_9_11 BOOLEAN)
(CONSTRAINTS '((=> _r_3_9_10 _r_3) (=> _r_3_9_11 _r_3) (OR (_r_1_5) (NOT _r_3_9_11))))
```

Fig. 6. The transformer engine converts a feature model to a logical web form description for PLATO. While the description shown above is Lisp-like, PLATO is scheduled to support a purely XML input format in the near future.

<input checked="" type="checkbox"/> Language	<input checked="" type="checkbox"/> Simple Arithmetic
	<input type="checkbox"/> Conditional
<input checked="" type="checkbox"/> Evaluator	<input checked="" type="checkbox"/> Evaluation semantics
	<input type="checkbox"/> Machine semantics
<input type="checkbox"/> Optimizer	<input type="checkbox"/> Constant folding
	<input checked="" type="checkbox"/> Short-circuit

Fig. 7. The SMARTFORM highlights two errors: 1) the Short – circuit feature requires the Conditional feature; 2) the Short – circuit feature cannot be solely selected without selection of its parent feature.

errors as illustrated in Figure 7.

IV. CONCLUSION

In this paper, we presented a web-based feature configuration tool that facilitates the process of feature selection. The user-interface produced by our tool includes feature validation implemented entirely in the browser that executes each time a user changes their selection, giving users real-time feedback.

In the future, we plan to extend SMARTFORM to enable tens, hundreds, or even thousands of users to collaboratively make a single feature selection. We hope to support a broad range of well-understood methodologies for collaborative feature selection, e.g. multiple views or multi-staging [11], [5]. Two of SMARTFORM's features were put in place to make extensions that support collaboration straightforward.

First, SMARTFORM was designed as a web application. By deploying on the web, users all over the world can participate in the feature selection process almost immediately because they only need access to a standard web browser and Internet connection. As an added benefit, because they are using web browsers, users are already familiar with basic operations and will become comfortable far more quickly than they would with custom software. Additionally, we as implementors can utilize a plethora of tools and methodologies designed to simplify the maintenance and construction of massively parallel applications.

Second, SMARTFORM produces web forms that allow users to make feature selections that violate the feature model (but inform them of the mistake). Such an interface aids collaboration because it is capable of representing the joint feature selection of numerous users. When collaboratively making a feature selection, a user might want to start from

scratch, or she might want start from the current joint progress of all her colleagues. Ideally, she would use the same interface in either case, but that requires the interface to represent disagreements among collaborators, which are violations of the feature model. Thus, by allowing an individual user to violate the feature model (at least temporarily), our tool paves the way for collaboration.

REFERENCES

- [1] Michal Antkiewicz and Krzysztof Czarnecki. FeaturePlugin: feature modeling plug-in for eclipse. In *OOPSLA workshop on eclipse technology exchange*, New York, NY, USA, 2004.
- [2] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the International Software Product Line Conference*, pages 7–20, 2005.
- [3] Wonseok Chae and Matthias Blume. Building a family of compilers. In *Proceedings of the 12th International Software Product Line Conference*, 2008.
- [4] Wonseok Chae and Matthias Blume. Language support for feature-oriented product line engineering. In *Proceedings of the First International Workshop on Feature-Oriented Software Development*, pages 3–10, 2009.
- [5] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.
- [6] Abdelrahman Osman Elfaki, Somnuk Phon-Amnuaisuk, and Chin Kuan Ho. Knowledge based method to validate feature models. In *Proceedings of the International Software Product Line Conference*, 2008.
- [7] Timothy Hinrichs, Jui-Yi Kao, and Michael Genesereth. Automatic web form construction via paraconsistent compilation to relational databases. Technical report, University of Chicago, 2010.
- [8] Mikolas Janota and Joseph Kiniry. Reasoning about feature models in higher-order logic. In *Proceedings of the 11th International Software Product Line Conference*, pages 13–22, 2007.
- [9] Mikołaj Janota, Goetz Botterweck, Radu Grigore, and Joao Marques-Silva. How to complete an interactive configuration process? In *Proceeding of 36th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, 2010.
- [10] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euseob Shin, and Moonhang Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.
- [11] Kwanwoo Lee, Kyo Chul Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *Proceedings of the 7th International Conference on Software Reuse*, pages 62–77, 2002.
- [12] Mike Mannion. Using first-order logic for product line model validation. In *Proceedings of the Second International Conference on Software Product Lines*, pages 176–187, 2002.
- [13] Workshop on Scalable Modeling Techniques for Software Product Lines. <http://kishi-www.jaist.ac.jp/SCALE2009/>, 2009.
- [14] Software Product Lines Online Tools. <http://www.splot-research.org/>, 2009.
- [15] Jules White, Douglas C. Schmidt, Egon Wuchner, and Andrey Nechypurenko. Automating product-line variant selection for mobile devices. In *Proceedings of the International Software Product Line Conference*, pages 129–140, 2007.

Previously published ICB - Research Reports

2009

No 36 (December 2009)

Stefan Strecker: Ein Kommentar zur Diskussion um Begriff und Verständnis der IT-Governance – Anregungen zu einer kritischen Reflexion

No 35 (August 2009)

Irene Rüngeler, Michael Tüxen, Erwin P. Rathgeb: Considerations on Handling Link Errors in SCTP

No 34 (June 2009)

Dimka Karastoyanova, Raman Kazhamiakin, Andreas Metzger, Marco Pistore (Eds.): Workshop on Service Monitoring, Adaptation and Beyond

No 33 (May 2009)

Heimo Adelsberger, Andreas Drechsler, Tobias Bruckmann, Peter Kalvelage, Sophia Kinne, Jan Pellinger, Marcel Rosenberger, Tobias Trepper: Einsatz von Social Software in Unternehmen – Studie über Umfang und Zweck der Nutzung

No 32 (April 2009)

Barth, Manfred; Gadatsch, Andreas; Kütz, Martin; Rüdiger, Otto; Schauer, Hanno; Strecker, Stefan: Leitbild IT-Controller/-in – Beitrag der Fachgruppe IT-Controlling der Gesellschaft für Informatik e. V.

No 31 (April 2009)

Frank, Ulrich; Strecker, Stefan: Beyond ERP Systems: An Outline of Self-Referential Enterprise Systems – Requirements, Conceptual Foundation and Design Options

No 30 (February 2009)

Schauer, Hanno; Wolff, Frank: Kriterien guter Wissensarbeit – Ein Vorschlag aus dem Blickwinkel der Wissenschaftstheorie (Langfassung)

No 29 (January 2009)

Benavides, David; Metzger, Andreas; Eisenecker, Ulrich (Eds.): Third International Workshop on Variability Modelling of Software-intensive Systems

2008

No 28 (December 2008)

Goedicke, Michael; Striewe, Michael; Balz, Moritz: „Computer Aided Assessments and Programming Exercises with JACK“

No 27 (December 2008)

Schauer, Carola: “Größe und Ausrichtung der Disziplin Wirtschaftsinformatik an Universitäten im deutschsprachigen Raum - Aktueller Status und Entwicklung seit 1992“

No 26 (September 2008)

Milen, Tilev; Bruno Müller-Clostermann: “ CapSys: A Tool for Macroscopic Capacity Planning“

No 25 (August 2008)

Eicker, Stefan; Spies, Thorsten; Tschersich, Markus: “Einsatz von Multi-Touch beim Softwaredesign am Beispiel der CRC Card-Methode“

No 24 (August 2008)

Frank, Ulrich: *"The MEMO Meta Modelling Language (MML) and Language Architecture – Revised Version"*

No 23 (January 2008)

Sprenger, Jonas; Jung, Jürgen: *"Enterprise Modelling in the Context of Manufacturing – Outline of an Approach Supporting Production Planning"*

No 22 (January 2008)

Heymans, Patrick; Kang, Kyo-Chul; Metzger, Andreas, Pohl, Klaus (Eds.): *"Second International Workshop on Variability Modelling of Software-intensive Systems"*

2007

No 21 (September 2007)

Eicker, Stefan; Annett Nagel; Peter M. Schuler: *"Flexibilität im Geschäftsprozess-management-Kreislauf"*

No 20 (August 2007)

Blau, Holger; Eicker, Stefan; Spies, Thorsten: *"Reifegradüberwachung von Software"*

No 19 (June 2007)

Schauer, Carola: *"Relevance and Success of IS Teaching and Research: An Analysis of the ‚Relevance Debate‘"*

No 18 (May 2007)

Schauer, Carola: *"Rekonstruktion der historischen Entwicklung der Wirtschaftsinformatik: Schritte der Institutionalisierung, Diskussion zum Status, Rahmenempfehlungen für die Lehre"*

No 17 (May 2007)

Schauer, Carola; Schmeing, Tobias: *"Development of IS Teaching in North-America: An Analysis of Model Curricula"*

No 16 (May 2007)

Müller-Clostermann, Bruno; Tilev, Milen: *"Using G/G/m-Models for Multi-Server and Mainframe Capacity Planning"*

No 15 (April 2007)

Heise, David; Schauer, Carola; Strecker, Stefan: *"Informationsquellen für IT-Professionals – Analyse und Bewertung der Fachpresse aus Sicht der Wirtschaftsinformatik"*

No 14 (March 2007)

Eicker, Stefan; Hegmanns, Christian; Malich, Stefan: *"Auswahl von Bewertungsmethoden für Softwarearchitekturen"*

No 13 (February 2007)

Eicker, Stefan; Spies, Thorsten; Kahl, Christian: *"Softwarevisualisierung im Kontext serviceorientierter Architekturen"*

No 12 (February 2007)

Brenner, Freimut: *"Cumulative Measures of Absorbing Joint Markov Chains and an Application to Markovian Process Algebras"*

No 11 (February 2007)

Kirchner, Lutz: "Entwurf einer Modellierungssprache zur Unterstützung der Aufgaben des IT-Managements – Grundlagen, Anforderungen und Metamodell"

No 10 (February 2007)

Schauer, Carola; Strecker, Stefan: "Vergleichende Literaturstudie aktueller einführender Lehrbücher der Wirtschaftsinformatik: Bezugsrahmen und Auswertung"

No 9 (February 2007)

Strecker, Stefan; Kuckertz, Andreas; Pawlowski, Jan M.: "Überlegungen zur Qualifizierung des wissenschaftlichen Nachwuchses: Ein Diskussionsbeitrag zur (kumulativen) Habilitation"

No 8 (February 2007)

Frank, Ulrich; Strecker, Stefan; Koch, Stefan: "Open Model - Ein Vorschlag für ein Forschungsprogramm der Wirtschaftsinformatik (Langfassung)"

2006

No 7 (December 2006)

Frank, Ulrich: "Towards a Pluralistic Conception of Research Methods in Information Systems Research"

No 6 (April 2006)

Frank, Ulrich: "Evaluation von Forschung und Lehre an Universitäten – Ein Diskussionsbeitrag"

No 5 (April 2006)

Jung, Jürgen: "Supply Chains in the Context of Resource Modelling"

No 4 (February 2006)

Lange, Carola: "Development and status of the Information Systems / Wirtschaftsinformatik discipline: An interpretive evaluation of interviews with renowned researchers, Part III – Results Wirtschaftsinformatik Discipline"

2005

No 3 (December 2005)

Lange, Carola: "Development and status of the Information Systems / Wirtschaftsinformatik discipline: An interpretive evaluation of interviews with renowned researchers, Part II – Results Information Systems Discipline"

No 2 (December 2005)

Lange, Carola: "Development and status of the Information Systems / Wirtschaftsinformatik discipline: An interpretive evaluation of interviews with renowned researchers, Part I – Research Objectives and Method"

No 1 (August 2005)

Lange, Carola: „Ein Bezugsrahmen zur Beschreibung von Forschungsgegenständen und -methoden in Wirtschaftsinformatik und Information Systems“

Research Group	Core Research Topics
Prof. Dr. H. H. Adelsberger Information Systems for Production and Operations Management	E-Learning, Knowledge Management, Skill-Management, Simulation, Artificial Intelligence
Prof. Dr. P. Chamoni MIS and Management Science / Operations Research	Information Systems and Operations Research, Business Intelligence, Data Warehousing
Prof. Dr. F.-D. Dorloff Procurement, Logistics and Information Management	E-Business, E-Procurement, E-Government
Prof. Dr. K. Echtle Dependability of Computing Systems	Dependability of Computing Systems
Prof. Dr. S. Eicker Information Systems and Software Engineering	Process Models, Software-Architectures
Prof. Dr. U. Frank Information Systems and Enterprise Modelling	Enterprise Modelling, Enterprise Application Integration, IT Management, Knowledge Management
Prof. Dr. M. Goedicke Specification of Software Systems	Distributed Systems, Software Components, CSCW
Prof. Dr. T. Kollmann E-Business and E-Entrepreneurship	E-Business and Information Management, E-Entrepreneurship/ E-Venture, Virtual Marketplaces and Mobile Commerce, Online Marketing
Prof. Dr. B. Müller-Clostermann Systems Modelling	Performance Evaluation of Computer and Communication Systems, Modelling and Simulation
Prof. Dr. K. Pohl Software Systems Engineering	Requirements Engineering, Software Quality Assurance, Software-Architectures, Evaluation of COTS/Open Source-Components
Prof. Dr.-Ing. E. Rathgeb Computer Networking Technology	Computer Networking Technology
Prof. Dr. A. Schmidt Pervasive Computing	Pervasive Computing, Ubiquitous Computing, Automotive User Interfaces, Novel Interaction Technologies, Context-Aware Computing
Prof. Dr. R. Unland Data Management Systems and Knowledge Representation	Data Management, Artificial Intelligence, Software Engineering, Internet Based Teaching
Prof. Dr. S. Zelewski Institute of Production and Industrial Information Management	Industrial Business Processes, Innovation Management, Information Management, Economic Analyses