Jan Bosch
Jaejoon Lee (Eds.)

LNCS 6287

Software Product Lines: Going Beyond

14th International Conference, SPLC 2010
Jeju Island, South Korea, September 2010
Proceedings

Springer

Lecture Notes in Computer Science 6287

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Jan Bosch Jaejoon Lee (Eds.)

Software Product Lines: Going Beyond

14th International Conference, SPLC 2010
Jeju Island, South Korea, September 13-17, 2010
Proceedings

Springer

Volume Editors

Jan Bosch
Intuit
Mountain View, CA, USA
E-mail: jan@janbosch.com

Jaejoon Lee
School of Computing and Communications
Lancaster University
Lancaster, UK
E-mail: j.lee@comp.lancs.ac.uk

Library of Congress Control Number: 2010933526

CR Subject Classification (1998): H.4, C.2, H.3, D.2, H.5, J.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-15578-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-15578-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Welcome from the General Chair

Welcome to the proceedings of SPLC 2010 which was held on the beautiful island of Jeju. It was the second time the International Software Product Line Conference (SPLC) came to Asia, where the IT-related industry is a key motive power for economic development and prosperity.

The current trend of globalization and rapid movement toward an IT-embedded society is pressuring industries to explore ways to meet the diverse needs of the global market most effectively and efficiently. Over the last decade or so, the field of software product lines has emerged as one of the most promising software development paradigms in drastically increasing the productivity of IT-related industries, and the product line community has grown and is still growing rapidly. The engineering discipline of software product lines has emerged as the "power for competitive advantage."

SPLC is the most prestigious and leading forum for researchers, practitioners, and educators in the field. SPLC 2010 provided a venue to the community for exchanging, sharing, and learning technologies and industrial experiences. The conference featured research and experience papers, tutorials, workshops, panels, a doctoral symposium, and demonstrations.

Creation of this outstanding technical program was not possible without the contribution of many people. I sincerely thank the authors, poster presenters, doctoral student session participants, workshop organizers and participants, tutorial presenters, keynote speakers, and panel participants who came from all over the world to share their research work, practical experiences, and valuable insights and knowledge with others.

In preparation for this conference, many people contributed their time and efforts and worked diligently over a year. To all members of the SPLC 2010 team, "thank you very much!" My special thanks go to the Software Engineering Society under the Korean Institute of Information Scientists and Engineers, and the industrial sponsors for their generous financial support and donations. Without this support, holding this conference may not have been possible.

Finally, I thank the conference attendees for their attendance and participation.

Kyo-chul Kang

Welcome from the Program Chairs

Since its rise to general awareness and popularity starting close to two decades ago, the concept of software product lines has take the center stage in the software reuse community. After more than four decades of research into effective and efficient reuse of software inside the four walls of the organization, and countless initiatives, software product lines presented an approach that has proven to provide real productivity improvements in the development cost of software-intensive products. This has allowed companies to increase their product portfolio with an order of magnitude, to allow for much higher degrees of configurability by customers, facilitated common look-and-feel across a wide product population and enabled companies to be more innovative by decreasing the cost of new product experiments. It achieved this by broadening the scope of study from technology to include process, business strategy and organizational aspects. Successful product lines address all aspects relevant to the organization and then adopt and institutionalize the approach in the company.

The software product line community is one where the collaboration between research and industry has been particularly strong and this has been one of its key success factors. The 14th Software Product Line Conference (SPLC 2010) represented the latest instance of a conference series that has developed into an institution and the premier meeting place for the community. This year's conference was based in Asia, the second time in the history of the conference series. Earlier instances took place in Europe and the USA. As the conference participation originates from all regions of the world, we are proud to be a truly global conference.

This year's conference received a high number of submissions, especially considering the economic realities that individuals, organizations and countries have been faced with. We received 90 full paper submissions and accepted 28 top-quality papers, giving an acceptance rate of 31%. The accepted papers cover various areas of software product line engineering including product line contexts, variability management, formal approaches, product validation, and feature modeling.

As we received many papers that we were unable to accept as full papers, but that we felt contained very valuable novel ideas, we introduced the notion of short papers. We invited 10 full papers as short ones and the authors had opportunities to present their ideas at the conference. In addition, we had 24 short paper submissions and accepted 4 short papers. Finally, we invited six posters from the short paper submissions.

Not surprising for a maturing discipline, the focus of the research papers is shifting from initial adoption of a software product line to the optimal use and evolution of an established product line. Consequently, research around features, including the evolution of feature models and the linking of feature models to code artifacts, is well represented. Software variability management is a second topic that is studied by several authors. However, in addition to the research on established topics, authors also reach beyond and into the future, including the application of the technology in new domains, such as the safety-critical, mobile and enterprise domains.

Continuously reinventing and reimagining the topic of software product lines is extremely important to ensure the continued viability and relevance of the research and practice communities and hence we would like to encourage the community to steer their curiosity, experiences and energy to this.

Concluding, as Program Chairs, we were honored to stand on the shoulders of those that held this position in the past and hope our efforts to serve the community in this capacity were appreciated by the participants in the SPLC 2010 conference.

Jan Bosch
Jaejoon Lee

Organization

General Chair	Kyo-chul Kang, (POSTECH, South Korea)
Program Co-chairs	Jan Bosch (Intuit, Inc., USA) Jaejoon Lee (Lancaster University, UK)
Industry Track Chair	Steve Livengood (Samsung Information Systems America, USA)
Panel Chair	Klaus Schmid (University of Hildesheim, Germany)
Workshop Chair	Goetz Botterweck (Lero, University of Limerick, Ireland)
Tutorial Chair	Liam O'Brien (NICTA, Australia)
Doctoral Symposium Chair	Tomoji Kishi (Waseda University, Japan)
Demonstration and Tools Chair	Stan Jarzabek (National University of Singapore, Singapore)
Hall of Fame Chair	David Weiss (Iowa State University, USA)
Publicity Chairs	Patrick Donohue (SEI, Carnegie Mellon University, USA) Hyesun Lee (POSTECH, South Korea)
Local Chair	Kwanwoo Lee (Hansung University, South Korea)

Program Committee

Full Papers

Eduardo Santana de Almeida	Federal University of Bahia and RiSE, Brazil
M. Ali Babar	IT University of Copenhagen, Denmark
Don Batory	University of Texas at Austin, USA
David Benavides	University of Seville, Spain
Goetz Botterweck	Lero, Ireland
Anders Caspár	Ericsson Software Research, Sweden
Paul Clements	Software Engineering Institute, USA
Sholom Cohen	Software Engineering Institute, USA
Davide Falessi	University of Rome Tor Vergata, Italy
Stuart Faulk	University of Oregon, USA
John Favaro	INTECS, Italy
Bill Frakes	Virginia Tech, USA
Alessandro Garcia	PUC-Rio, Brazil
Svein Hallsteinsen	SINTEF, Norway
Øystein Haugen	SINTEF, Norway
Patrick Heymans	University of Namur, PReCISE, Belgium
Stan Jarzabek	National University of Singapore, Singapore
Isabel John	Fraunhofer IESE, Germany
Tomoji Kishi	Waseda University, Japan
Kwanwoo Lee	Hansung University, South Korea
Frank van der Linden	Philips Healthcare, The Netherlands
Mikael Lindvall	Fraunhofer Center for Experimental Software Engineering, USA
Robyn Lutz	Iowa State University and Jet Propulsion Lab, USA
Tomi Männistö	Aalto University, Finland
John D. McGregor	Clemson University, USA
Hong MEI	Peking University, China
Maurizio Morisio	Politecnico di Torino, Italy
Dirk Muthig	Lufthansa Systems, Germany
Liam O'Brien	NICTA, Australia
Rob van Ommering	Philips Research, The Netherlands
Eila Ovaska	VTT Technical Research Centre of Finland, Finland
Klaus Pohl	University of Duisburg-Essen, Germany
Jeffrey Poulin	Lockheed Martin Systems Integration- Owego, USA
Juha Savolainen	Nokia, Finland
Klaus Schmid	University of Hildesheim, Germany
Steffen Thiel	Furtwangen University of Applied Sciences, Germany
Tim Trew	NXP Semiconductors, The Netherlands
David M. Weiss	Iowa State University, USA
Claudia Maria Lima Werner	Federal University of Rio de Janeiro, Brazil
Jon Whittle	Lancaster University, UK

Short Papers and Posters

Davide Falessi	University of Rome Tor Vergata, Italy
Dharmalingam Ganesan	Fraunhofer Center for Experimental Software Engineering, USA
Alessandro Garcia	PUC-Rio, Brazil
Patrick Heymans	University of Namur, PReCISE, Belgium
Isabel John	Fraunhofer IESE, Germany
Lawrence Jones	Software Engineering Institute, USA
Tomoji Kishi	Waseda University, Japan
Kwanwoo Lee	Hansung University, South Korea
John D. McGregor	Clemson University, USA
Dirk Muthig	Lufthansa Systems, Germany
Natsuko Noda	NEC Servie Platforms Research Laboratories, Japan
Pete Sawyer	Lancaster University, UK

Additional Reviewers

Vander Alves
Hamid Abdul Basit
Nelly Bencomo
Marco Eugênio Madeira Di Beneditto
Quentin Boucher
Rafael Capilla
Elder Cirilo
Andreas Classen
Chessman Correa
Deepak Dhungana
Jose Angel Galindo Duarte
Holger Eichelberger
Thomas Forster
Dharmalingam Ganesan
Nicolas Genon
Roy Grønmo
Zhang Hongyu
Arnaud Hubaux
Martin Fagereng Johansen
Heng Boon Kui
Uirá Kulesza
Kim Lauenroth
Fabiana Marinho
Octavio Martin-Diaz
Raphael Michel
Varvana Myllärniemi
Natsuko Noda
Camila Nunes
Gøran K. Olsen
Hannu Peltonen
Xin Peng
Gilles Perrouin
Andreas Pleuss
Mikko Raatikainen
Fabricia Roos
Rodrigo Santos
Germain Saval
Vanessa Stricker
Andreas Svendsen
Eldanae Teixeira
Juha Tiihonen
Federico Tomassetti
Marco Torchiano
Rayner Vintervoll
Andrzej Wasowski
Xue Yinxing
Xiaorui Zhang
Wei Zhang

Organization and Sponsors

Organization

Sponsors

Platinum Level Sponsors

Gold Level Sponsors

Silver Level Sponsors

Table of Contents

Product Line Context

Formal Approaches

Experience Papers

Variability Management

Product Validation 1

Product Validation 2

Feature Modeling

Examples of Product Lines

MDA and Business Context

Short Papers

Posters

Panel Overviews

Tutorial Overviews

Workshop Overviews

Context-Dependent Product Line Practice for Constructing Reliable Embedded Systems

Naoyasu Ubayashi[1], Shin Nakajima[2], and Masayuki Hirayama[3]

[1] Kyushu University, Japan
[2] National Institute of Informatics, Japan
[3] Software Engineering Center, Japan
ubayashi@acm.org, nkjm@nii.ac.jp, m-hiraya@ipa.go.jp

Abstract. This paper proposes a new style of product line engineering methods. It focuses on constructing embedded systems that take into account the contexts such as the external physical environments. In the current product line engineering, the feature analysis is mainly conducted from the viewpoint of system configurations: how hardware and software components are configured to constitute a system. In most cases, contexts are not considered explicitly. As a result, unexpected and unfavorable behavior might emerge in a system if a developer does not recognize any possible conflicting combinations between the system and contexts. To deal with this problem, this paper provides the notion of a context-dependent product line, which is composed of the system and context lines. The former is obtained by analyzing a family of systems. The latter is obtained by analyzing features of contexts associated to the systems. In the proposed method, each feature is described using VDM++. The configuration of selected system components and contexts can readily be checked with VDM++ Toolset.

1 Introduction

This paper proposes a new style of product line engineering (PLE) [2] method for constructing reliable embedded systems that take into account the contexts such as the external physical environments. Many of the embedded systems not only affect their environments through actuators but also are affected by their environments through sensors. In this paper, the term *context* refers to the real world such as the usage environments that affect the system behavior. It is important to provide a context-dependent development method for constructing safe and reliable systems. Although all of the embedded systems are not necessarily context-dependent, many of the consumer appliances such as mobile phone, air conditioner, and car electronics are context-dependent. If fatal defects are included in these products, the large-scale recall is not avoidable. The goal of this paper is to improve the reliability of such kinds of embedded systems.

PLE is a promising approach to developing embedded systems. In PLE, a product is constructed by assembling core assets, components reused in a family of products. These core assets are identified by analyzing characteristics needed in a family of products. This activity is called the feature analysis [14].

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 1–15, 2010.

In the current PLE, the feature analysis is mainly conducted from the viewpoint of system configurations: how hardware and software components are configured to construct a system—the contexts are not considered explicitly in most cases. As a result, unexpected and unfavorable behavior might emerge in a system if a developer does not recognize any possible conflicting combinations between the system and contexts. This behavior might cause a crucial accident. It, however, is not easy to detect this behavior only by reviewing each of system and context requirements because this unfavorable behavior emerges through incidental combinations of a system and contexts. It is important to detect the unfavorable behavior systematically at the early stage of the development.

To deal with the above problem, this paper proposes the notion of a context-dependent PLE in which a product line is divided into two kinds of lines: the system line and the context line. The former is a line obtained by analyzing the features of hardware and software components that consist of a family of systems. The latter is obtained by analyzing the features of contexts.

In the proposed method, each feature description is specified using VDM++ [4], a language for lightweight formal approaches in which formal methods are used as a tool for describing the essential aspects of systems rigorously. VDM++ is an object-oriented extension of VDM-SL (The Vienna Development Method – Specification Language) [5], a formal specification language for the rigorous software modeling. VDM++ is one of the most popular formal methods in Japan. The *Mobile Felica Chip* project in Japan developed a very large embedded system using VDM++. In this project [18], the specification included over 100,000 lines of VDM++. For such a reason, we adopted VDM++ for feature descriptions. The correctness of the configuration of selected hardware components, software components, and contexts can be formally validated at the specification level by using VDMTools [21], a tool for supporting VDM++.

The remainder of this paper is structured as follows. In Section 2, problems in the current PLE is pointed out. In Section 3, the context-dependent PLE method is introduced to deal with the problems. In Section 4, a method for describing the core asset specifications and validating them is provided. In Section 5, related work is introduced. Concluding remarks are provided in Section 6.

2 Motivation

In this section, typical disadvantages in the current PLE are pointed out. The necessity of introducing the notion of contexts is claimed by describing the specification of an electric pot as an example—its context is the water.

2.1 Example — An Electric Pot

An electric pot is an embedded system for boiling the water. Here, for simplicity, only the following is considered: 1) the pot has three hardware components including a heater, a thermistor, and a water level sensor; 2) the pot controls the water temperature by turning on or off the heater; 3) the pot changes its

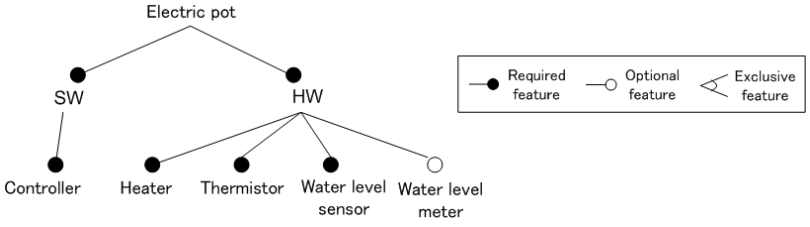


Fig. 1. Feature analysis for an electric pot

mode to the heat-retaining mode when the temperature becomes 100 Celsius; and 4) the pot observes the volume from the water level sensor that detects whether the water is below or above a certain base level.

2.2 Requirement Specification Process in PLE

In order to make the early stage of the software development systematically, the feature-oriented modeling (FOM) [14] has been proposed especially for embedded systems, which is aimed to support PLE. Since, for example, a consumer appliance has a wide variety of similar but different products, its development requires to identify a set of commonalities and variabilities. Ideally, a product is developed by assembling all the commonalities and some subset of variabilities for fulfilling the requirements of the very product. FOM is one of the modeling method used in PLE.

FOM provides a tree notation to explicitly represent the relationships among the identified features; some are commonalities and the others variabilities. The features are arranged in a tree where a lower node becomes a constituent of the upper one. Figure 1 is a portion of the feature tree for the example electric pot. The top node, `Electric pot` is decomposed into `SW` and `HW`. `HW` is, in turn, expanded into three mandatory system assets (`Heater`, `Thermistor`, and `Water level sensor`) and an optional asset (`Water level meter`).

The features that can be commonly reused in multiple products are accumulated as core assets. Reusing core assets of high quality can not only improve the quality of each product but also shorten the development time. However, if a core asset includes a fatal defect, it spreads to all of the products constructed from the same product line. Quality is a key factor of PLE practices.

The following is a typical process for constructing the requirement specifications based on the traditional PLE. This process consists of three steps.

1) Analyze features: First, a feature tree is created by analyzing the features of a product family, and requirement of each feature is specified. A specification can be reused if it is accumulated as a core asset. Here, as an example, let us consider the specification of the `Controller` feature. In most cases, this specification is described by implicitly taking into account the specific contexts—for example, such a context that the water is boiled under the normal air pressure. A developer describes the software logic corresponding to the specific contexts—in

this case, the pot continues to turn on a heater switch until the water temperature becomes 100 Celsius. Below is the specification described in VDM++. The first line is the signature of the `Boil` function that has no arguments and no returned value. This function describes that `Controller` continues to turn on `Heater` while the value of the temperature obtained from `Thermistor` is below 100 Celsius.

```
Boil: () ==> ()
Boil() ==
  while thermistor.GetTemperature() < 100.0
    do heater.On();
```

2) Select a set of features: Next, a set of features are selected from the feature tree. For example, one software feature `Controller` and three hardware features including `Heater`, `Thermistor`, and `Water level sensor` are selected for an electric pot with the minimal functions—all features are mandatory.

3) Validate a composed system specification: Lastly, the system requirement is validated by reviewing a set of feature specifications. The pot with the above `Boil` function behaves correctly under the normal circumstances.

2.3 Problems in Traditional Approach

Although this traditional PLE process is effective, there is a room for improvements in it because it does not explicitly consider the variability of contexts such as the water and air pressure. The above `Boil` specification seems to be correct. However, faults may occur if the expected contexts are changed—for example, the circumstance of the low air pressure. While this specification satisfies the requirements as an electric pot shown in 2.1, the conflict emerges between this specification and the changed context. Because the boiling point of the water is below 100 Celsius under the circumstance of the low air pressure, the software controller continues to heat the water even if its temperature becomes the boiling point. As a result, the water evaporates and finally its volume will be empty. The water level sensor observes the volume, and the pot stops heating. Although this behavior satisfies the above system specification, the pot might be useless for those who climb high mountains where the air pressure is low and use the pot there. If a developer considers those people as customers of the pot, the above behavior is regarded as a system failure. It is difficult to detect this kind of defects because the specifications concerning contexts tend to be tangled and crosscutting over multiple feature specifications.

It is not easy to deal with this kind of problem using a feature analysis tree composed of only hardware and software features. Most of developers would face the following problems: it is not easy to reuse the specifications because the system logic tends to be specified without clarifying the existence of contexts; many features might be modified whenever expected contexts are changed; it is not easy to keep the consistency among the features because they are modified frequently; and it is not easy to validate a product specification as a whole

because certain unexpected system behavior might emerge due to the conflict between the system and the changed contexts.

2.4 Our Approach

To deal with these problems, this paper proposes the context-dependent PLE method with lightweight formal approaches. This method has the following characteristics: 1) A product line is explicitly separated into two kinds of lines –the system line and the context line. The former is represented by a system feature tree, and the latter is represented by a context feature tree; 2) The specification of each feature is accumulated as a core asset if the feature is appeared in multiple products. Not only system but also context feature specifications can be reused as core assets; and 3) These specifications are described in a lightweight formal language. A product specification composed of the selected feature specifications is validated using tools that support the language.

Formal methods are mathematically rigorous techniques for the specification, design, and verification. Traditional formal methods such as VDM and Z have been used in the development of dependable systems such as railway systems and nuclear power plants that require high reliability, safety, and security. It is meaningful to apply formal methods to PLE that requires high dependability. Although formal methods are effective, the adoption of full formalization that needs mathematical proof is difficult and expensive. As an alternative approach, the notion of lightweight formal approaches has been proposed.

Lightweight formal approaches are categorized into *formal methods lite* [12] and *lightweight formal methods* [9]. The former encourages a developer to use formal methods as a tool for describing the essential aspects of a system rigorously, and the latter aims to the automated verification. VDMTools, a *formal method lite* supporting tool, provides facilities including syntax checking, type checking, proof obligation generation, and interpretive test execution.

3 Context-Dependent PLE

3.1 System Line and Context Line

Figure 2 illustrates a system line and a context line for an electric pot product family. The context feature tree represents the expected contexts that should be taken into account when each pot is used. For example, this feature tree consists of two kinds of contexts including `Air pressure` and `Liquid`. The `Air pressure` feature is expanded into three exclusive features including `High`, `Normal`, and `Low`. Similarly, the `Liquid` feature is expanded into `Water`, `Milk`, and so on.

Requirement specifications for each electric pot product can be constructed by selecting and combing features from system and context lines. When a developer specifies the requirements of a typical electric pot used in normal circumstances, the following features are selected: four system assets including `SW-Controller`, `HW-Heater`, `HW-Thermistor`, and `HW-Water level sensor`; and two context features including `Air pressure-Normal` and `Liquid-Water`.

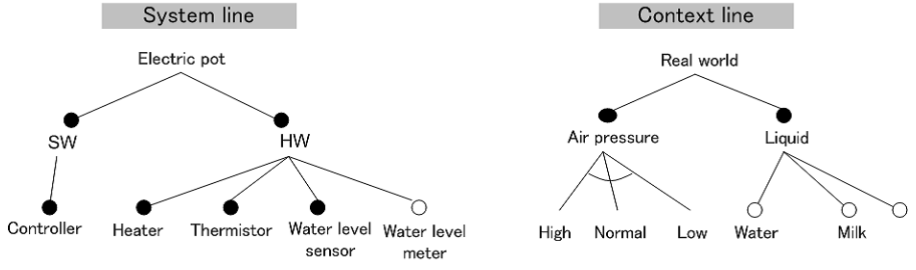


Fig. 2. System line and context line

3.2 Activities in Context-Dependent PLE

Our PLE framework consisting of core asset development and product development is called *context-dependent product line practice with lightweight formal approaches* (CD-PLP). The activities are as follows.

Core asset development: *1) System asset development:* analyze the system (hardware and software) features and develop the system assets used in a series of products. Each system asset is described formally using a lightweight formal language such as VDM++ and validated using lightweight formal tools such as VDMTools. *2) Context asset development:* analyze the context features and develop the context assets commonly used in a series of products. Each asset is described and validated using the same lightweight formal techniques.

Product development: *1) Definition of product configuration:* select system features required in a product from a set of system assets. *2) Definition of expected contexts:* select context features expected in a product from a set of context assets. *3) Validation of overall specifications:* validate whether unexpected behavior emerges due to the unexpected combination of contexts and product configurations. For this purpose, lightweight formal tools are used. If unfavorable behavior is detected, confirm the followings: whether selected asset specifications are really correct; and whether existing asset specifications need to be changed for adapting changed contexts.

3.3 Context Analysis Method

We provide a context analysis method for dividing a product line into system lines and context lines. The method is a key factor for practicing our approach.

Figure 3 illustrates the result of the context analysis. The upper and lower sides show a system and contexts, respectively. Hardware for observing and controlling contexts is located as a boundary that separates contexts from a system.

A UML profile shown in Table 1 is provided for context analysis. This profile can describe system elements, context elements, and association between them: three kinds of stereotypes including $\ll Context \gg$, $\ll Hardware \gg$, and $\ll Software \gg$ are defined as an extension of the UML class; and four kinds

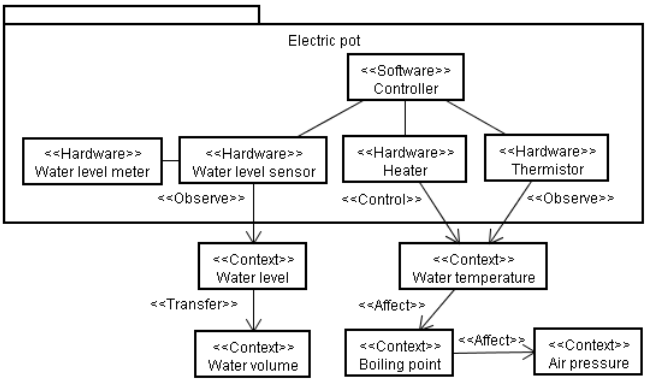


Fig. 3. Context analysis for an electric pot

Table 1. A UML profile for context analysis

Name	Category	Definition
$\ll Context \gg$	Class	Context element
$\ll Hardware \gg$	Class	Hardware element
$\ll Software \gg$	Class	Software element
$\ll Observe \gg$	Association	Hardware observes a context
$\ll Control \gg$	Association	Hardware controls a context
$\ll Transfer \gg$	Association	Data is transformed into different form because hardware cannot observe the original data directly
$\ll Affect \gg$	Association	Data from the target context is affected by other context

of stereotypes including $\ll Observe \gg$, $\ll Control \gg$, $\ll Transfer \gg$, and $\ll Affect \gg$ are defined as an extension of the UML association. The arrow of $\ll Observe \gg$ and $\ll Control \gg$ indicates the target of observation and control. The arrow of $\ll Affect \gg$ indicates the source of affect. The arrow of $\ll Transfer \gg$ indicates the source of transformation.

Steps for context analysis: A UML diagram shown in Figure 3 is created by the following procedure. First, the context elements ($\ll Context \gg$), which are observed ($\ll Observe \gg$) or controlled ($\ll Control \gg$) directly by hardware ($\ll Hardware \gg$), are extracted. In case of an electric pot, *water level* and *water temperature* are extracted since *water level* is observed by the water level sensor and *water temperature* is controlled by the heater. Next, impact factors that affect the states of these contexts elements are extracted using the following guide words [15], hints for deriving related elements: 1) factor that determines the upper limit; 2) factor that determines the lower limit; 3) factor related to a specific value; 4) factor that interferes the observation; and 5) factor that interferes the control. $\ll Affect \gg$ is used for this analysis. The *boiling point* can be extracted as an impact factor for the *water temperature* by applying the guide word *"factor that determines the upper limit"* since the temperature does

not become higher than the boiling point. Furthermore, the *air pressure* can be extracted as an impact factor for the *boiling point* by applying the guide word *"factor related to a specific value"* since the boiling point of the water is 100 Celsius under the circumstance of 1.0 atm. An element directly observed by hardware might be an alternative context in such a case that the hardware cannot observe the original value of the target context. For example, the pot wants to observe not the *water level* but the *water volume*. The $\ll Transfer \gg$ association is used in this situation.

Extraction of system assets: In Figure 3, there are system elements including controller (software), heater (hardware), thermistor (hardware), water level sensor (hardware), and water level meter. All of the elements are extracted as system assets consisting of a system line feature tree shown in Figure 2.

Extraction of context assets: To constitute a context line, context assets need to be extracted from Figure 3. In this case, however, all of the context elements are not necessarily considered as assets. The context assets can be extracted by the following procedure: 1) start from the context elements ($\ll Context \gg$) that are directly observed ($\ll Observe \gg$) or controlled ($\ll Control \gg$) by the hardware ($\ll Hardware \gg$); 2) trace the $\ll Transfer \gg$ associations; and 3) final context elements and their impact factors that are traced by $\ll Affect \gg$ associations. In case of an electric pot, *water volume*, *water temperature*, and *air pressure* are extracted as context elements to be considered. In Figure 3, valued elements are extracted as contexts. Although *water volume* and *water temperature* are extracted separately, they are properties of the *water*. In the context line, the *liquid*, generalization of the *water*, and *air pressure* are extracted as final context assets. *High*, *Normal*, and *Low* can be derived by analyzing the variation of *air pressure*.

4 Modeling and Validation Using VDM++

Based on CD-PLP, a method for describing asset specifications using VDM++ and validating them using VDMTools is demonstrated in this section.

4.1 Core Asset Development Using VDM++

Table 2 shows the VDM++ asset specifications for an electric pot product line.They are categorized into testing (`User-Test` and `RealWorld`), software (the naming convention is SYSTEM-SW-*asset name*), hardware (SYSTEM-HW-*asset name*), and contexts (CONTEXT-*asset name*). The appendix includes the full asset descriptions. The `UserTest` is a specification for validating composed product specifications under the circumstances specified by `RealWorld`. These test specifications are also regarded as the core assets because they can be reused at the validation phase. In our approach, not only system assets but also context assets are described based on functionality. For example, we think that the water has a function for *adding the temperature* when the water is heated.

Table 2. Category of asset specifications

Category	Name	Asset
Testing	UserTest	Test specification
	RealWorld	Context setting
System line (SW)	SYSTEM-SW-controller	Controller
System line (HW)	SYSTEM-HW-heater	Heater
	SYSTEM-HW-thermistor	Thermistor
	SYSTEM-HW-liquid-level-sensor	Level sensor
Context line	CONTEXT-atmospheric-air-pressureplace	Air pressure
	CONTEXT-atmospheric-air-pressureplace-high	High air pressure ($> 1.0atm$)
	CONTEXT-atmospheric-air-pressureplace-normal	Normal air pressure (1.0 atm)
	CONTEXT-atmospheric-air-pressureplace-low	Low air pressure ($< 1.0atm$)
	CONTEXT-liquid-water	Water

Figure 4 illustrates the relation among testing, software, hardware, and context specifications for an electric pot. For simplicity, only main specifications are shown in Figure 4. The `UserTest` invokes the software controller `SYSTEM-SW-controller`, setups the context `RealWorld` of the electric pot, and sends the command `Boil` to the controller. Below is the VDM++ code that specifies the logic for continuing to turn on the heater `SYSTEM-HW-heater` until the water temperature becomes 100 Celsius. The condition *"the volume of the pot must not be empty"* is also specified in the precondition and postcondition.

```
// Controller (software)
public
Boil: () ==> ()
Boil() ==
  while thermistor.GetTemperature() < 100.0 and
        liquid_level_sensor.IsOn() = true
    do heater.On()
pre  liquid_level_sensor.IsOn() = true
post liquid_level_sensor.IsOn() = true;
```

The heater heats the water `Context-liquid-water`.

```
// Heater (hardware)
public On: () ==> ()
On() ==
 (sw := <On>; realworld_liquid.AddTemperature());
```

In the water specification, only the physical phenomena—the water evaporates after its temperature becomes the boiling point—is described.

```
// Water (context) --phisical phenomena is described simply
public
AddTemperature: () ==> ()
AddTemperature() ==
  if temperature < boiling_point(aap.GetAtm())
     then  temperature := temperature + 1.0
     else (temperature := boiling_point(aap.GetAtm());
           amount := amount - 1.0  --- evaporation )
pre  temperature <= boiling_point(aap.GetAtm())
post temperature <= boiling_point(aap.GetAtm());
```

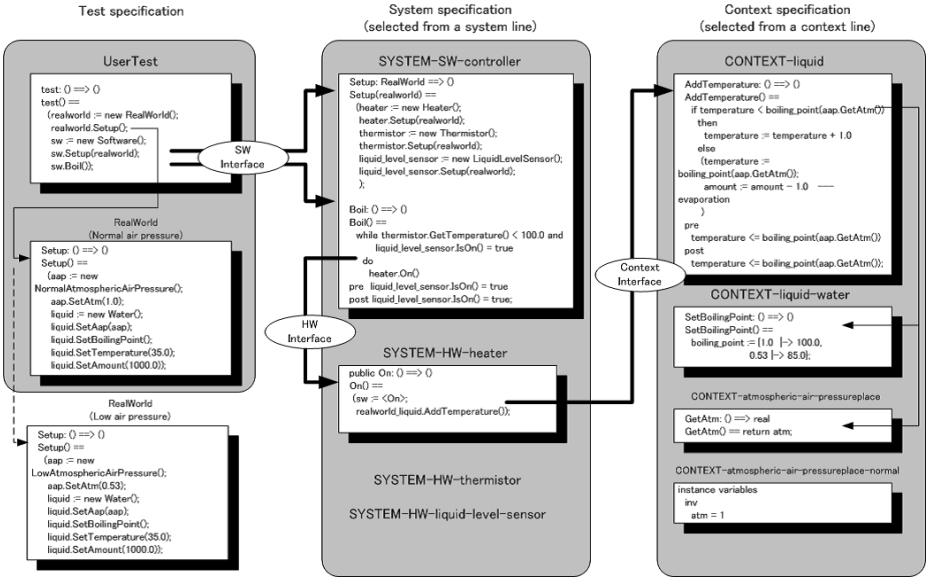


Fig. 4. Asset specifications described in VDM++

As illustrated in Figure 4, each VDM++ specification is described based on the principle of *separation of concerns*: the context specifications do not include the descriptions related to hardware but describe features only related to contexts themselves; the hardware specifications do not include the descriptions related to the software; and the software specifications do not include the descriptions related to testing. On the other hand, the hardware for observing and controlling the contexts can access the contexts through only the context interfaces. The test specifications and the software specifications access the software and the hardware through only software interfaces and hardware interfaces respectively in the same way.

4.2 Product Development Using VDMTools

Using core assets specified in 4.1, a developer can configure an actual electric pot used in the expected contexts. We call this pot `PotX`.

[Step1] Define product configuration: The specification of `PotX` that provides only the minimum functionality is configured by four core system assets including `SYSTEM-SW-controller`, `SYSTEM-HW-heater`, `SYSTEM-HW-thermistor`, and `SYSTEM-HW-liquid-level-sensor`.

[Step2] Define expected contexts: As the examples of expected contexts for `PotX`, we consider two kinds of contexts: A) the water is boiled under the circumstance of the normal air pressure; and B) the water is boiled under a certain of the air pressure lower than the normal. We call the former `ContextA` and the latter

`ContextB`, respectively. The specification of `ContextA` is configured by two core context assets including `CONTEXT-atmospheric-air-pressureplace-normal` and `CONTEXT-liquid-water`. On the other hand, the specification of `ContextB` is configured by two core context assets including `CONTEXT-atmospheric-air-pre-ssureplace-low` and `CONTEXT-liquid-water`.

[Step3] Validate overall specifications: Using VDMTools, a developer can validate overall specifications of two cases: *PotX + ContextA* and *PotX + ContextB*. The difference of these cases is which one is selected for `CONTEXT-atmo-spheric-air-pressureplace-normal` or `CONTEXT-atmospheric-air-pressur-eplace-low`. The validation of these two cases can be checked using the test execution facility provided by VDMTools.

Although the test execution in case of *PotX + ContextA* terminates normally—the pot stops boiling when the temperature of the water becomes 100 Celsius, the test execution in case of *PotX + ContextB* raises an error as shown in Figure 5—the precondition of the `Boil` operation is violated as follows.

```
post liquid_level_sensor.IsOn() = true;
[Message from VDM++ ToolBox]
Run-Time Error 59:
  The post-condition evaluated to false
```

The pre-/post-conditions of the `Boil` function means "the switch of the water level sensor should be on—the volume of the water must not be empty". This condition represents a favorable property needed for every electric pot. In case of *PotX + ContextB*, the pot continues to heat the water, and eventually all of the water evaporates—the precondition is violated. This test execution facility is effective for detecting unexpected behavior.

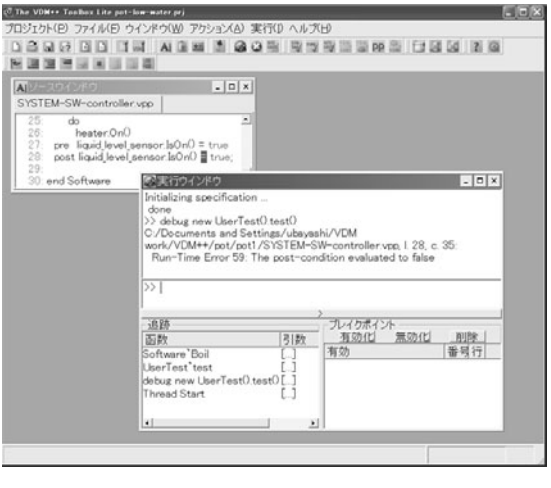


Fig. 5. Result of test execution (context B) [Japanese version]

When a testing finds defects emerged from the incidental combinations of a system and contexts, a developer should consider the following: how should system functions be configured ?—should new hardware components be added ? or should software specifications be modified ? The testing provides a developer an opportunity for reconsidering the roles of the hardware and software. In order to deal with the defects emerged in the *PotX + ContextB* case, an air pressure sensor should be added because the pot needs the boiling operation coping with changes in the air pressure.

The validation method shown here deals with only a brief example, and a developer might be aware of the possibility of defects when he or she takes into account the air pressure as an expected context. Someone might claim that this testing is not needed for validating specifications. However, in general, there are many contexts that should be taken into account. Moreover, the relations among these contexts are complicated. It is not easy for a developer to find defects emerged by the combinations of contexts.

The validation method proposed here does not make sense if a developer cannot understand that the air pressure is one of the expected contexts. To deal with this problem, we provided a method for extracting context assets as shown in 3.3. In the traditional PLE approaches, there was no way except accumulating domain knowledge.

5 Related Work

There are some studies that take into account the real world as a modeling target. For example, S.Greenspan et al. claim the necessity of introducing real world knowledge into requirement specifications [6]. M.Jackson proposes a *problem frame* [11] in which relations between machine (system) and the real world. The notion of contexts in this paper corresponds to the real world in the problem frame. This paper claimed the necessity of introducing the notion of contexts not only in the individual product development but also in the product line development. The adaptation of problem frames to product lines has been introduced in [22]. Examples of formalising requirements with problem frames can be found in [7].

C.Atkinson et al. propose a PLE method called KobrA [1] in which the context realization models are described by analyzing contexts of target systems. However, the systems and contexts are simultaneously described in KobrA. On the other hand, the system lines are completely separated from the context lines in our approach. We believe that our approach is effective comparing to the way in which contexts are taken into account as one of the system concerns. There are contexts features that can be shared among multiple system lines. If a context belongs to a specific system line, the context cannot be reused in other system lines.

K.C.Kang et al. propose a method for categorizing features into four layer including capability, operating environment, domain technology, and implementation technique [13]. Recently, they point out the importance of introducing the

viewpoint of *usage context*. The notion of context becomes a hot topic in the PLE research community. The method in this paper would be one of the first proposals for the systematic approach to the context-dependent PLE.

It is important to construct product lines from the viewpoint of system safety [15]. J.Dehlinger et al. propose a method for applying SFTA (Software Fault Tree Analysis) to product lines[3]. They also propose a method that integrates SFTA and SFMECA (Software Failure Modes, Effects, and Criticality Analysis). J.Liu et al. propose a method that applies SFTA and state based modeling to product line safety analysis [16]. However, contexts are not considered explicitly in these methods.

Although there are many case studies that apply formal methods to system descriptions, descriptions of contexts are not rarely formalized. It is also important to formalize the feature trees for contexts. P.Höfner et al. propose feature algebra [8] for formalizing features. J.Sun et al. propose a formal semantics and verification method for feature modeling [19]. A formal semantics is defined using the first-order logic, and it is validated using the Z/EVES theorem prover [17]. The consistency of a feature model and its configurations are verified by encoding the semantics into the Alloy Analyzer [10], a tool for Alloy—a simple structural modeling language based on the first-order logic.

The notion of contexts is similar to aspect orientation because a context tends to crosscut over system elements. To improve the expressiveness of contexts, we previously proposed AspectVDM[20], a VDM-based AOP language for describing crosscutting features as aspects.

6 Conclusions

This paper proposed a new PLE method that takes into account the contexts. We also provided a method for describing core asset specifications using VDM++ and validating them using VDMTools. However, only the functionality can be checked using the current our approach. To deal with this problem, we plan to apply other lightweight formal tools such as the Alloy analyzer to verifying the correctness of composing system and context features. We believe that our approach is the first step towards the context-dependent PLE.

References

1. Atkinson, C., et al.: Component-Based Product Line Engineering with the UML. Addison-Wesley, Reading (2001)
2. Clements, P., Northrop, L.: Software Product Lines. Addison-Wesley, Reading (2001)
3. Dehlinger, J., Lutz, R.: Software Fault Tree Analysis for Product Lines. In: Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE 2004), pp. 12–21 (2004)
4. CSK: VDMTools –The CSK VDM++ Language, http://www.vdmtools.jp/files/langmanpp_a4E.pdf

5. Fitzgerald, J., Larsen, G.P.: Modeling Systems, Practical Tools and Techniques in Software Development. Cambridge University Press, Cambridge (1998)
6. Greenspan, S., Mylopoulos, J., Borgida, A.: Capturing More World Knowledge in the Requirements Specification. In: Proceedings of International Conference on Software Engineering (ICSE 1982), pp. 225–234 (1982)
7. Hayes, I., Jackson, M., Jones, C.: Determining the specification of a control system from that of its environment. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 154–169. Springer, Heidelberg (2003)
8. Höfner, P., Khedri, R., Möller, B.: Feature Algebra. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 300–315. Springer, Heidelberg (2006)
9. Jackson, D., Wing, J.: Lightweight Formal Methods. IEEE Computer 29(4), 21–22 (1996)
10. Jackson, D.: Software Abstractions. The MIT Press, Cambridge (2006)
11. Jackson, M.: Problem Frame: Analyzing and Structuring Software Development Problems. Addison-Wesley, Reading (2001)
12. Jones, C.B.: A Rigorous Approach To Formal Methods. IEEE Computer 29(4), 20–21 (1996)
13. Kang, K.C., Kim, S., Lee, J., Shin, E., Huh, M.: FORM: A Feature-oriented Reuse Method with Domain-specific Reference Architecture. Annals of Software Engineering 5, 143–168 (1998)
14. Kang, K.C., Lee, J., Donohoe, P.: Feature-Oriented Product Line Engineering. IEEE Software 9(4), 58–65 (2002)
15. Leveson, N.G.: Safeware: System Safety and Computers. Addison-Wesley, Reading (1995)
16. Liu, J., Dehlinger, J., Lutz, R.: Safety Analysis of Software Product Lines Using State-Based Modeling. In: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005), pp. 21–30 (2005)
17. Saaltink, M.: The Z/EVES system. In: Bowen, J.P., Hinchey, M.G., Till, D. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 72–85. Springer, Heidelberg (1997)
18. Sahara, S.: Current status of VDMTools. Talk at the 2nd Overture Workshop, FM 2006, Hamilton (August 2006), http://www.overturetool.org/downloads/ows2/slides6.pdf
19. Sun, J., Zhang, H., Fang, Y., Wang, L.H.: Formal Semantics and Verification for Feature Modeling. In: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2005), pp. 303–312 (2005)
20. Ubayashi, N., Nakajima, S.: Context-aware Feature-Oriented Modeling with an Aspect Extension of VDM. In: Proceedings of the 22nd Annual ACM Symposium on Applied Computing (SAC 2007), pp. 1269–1274 (2007)
21. VDMTools, http://www.vdmtools.jp/
22. Zuo, H., Mannion, M., Sellier, D., Foley, R.: An Extension of Problem Frame Notation for Software Product Lines. In: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005), pp. 499–505 (2005)

Appendix

Test specification

```
class UserTest
instance variables
  realworld : RealWorld;
  sw : Software;
operations
  public test: () ==> bool
  test() ==
    (realworld := new RealWorld();
     realworld.Setup();
     sw := new Software(); sw.Setup(realworld); sw.Boil();
     return true);
end UserTest
```

RealWorld

```
-- Context A: normal air pressure
class RealWorld
instance variables
  public aap: NormalAtmosphericAirPressure;
  public liquid : Water;
operations
  public Setup: () ==> ()
  Setup() ==
    (aap := new NormalAtmosphericAirPressure();
     aap.SetAtm(1.0);
     liquid := new Water();
     liquid.SetAap(aap);
     liquid.SetBoilingPoint();
     liquid.SetTemperature(35.0); liquid.SetAmount(1000.0));
end RealWorld

-- Context B: air pressure lower than the normal
class RealWorld
instance variables
  public aap: LowAtmosphericAirPressure;
  public liquid : Water;
operations
  public Setup: () ==> ()
  Setup() ==
    (aap := new LowAtmosphericAirPressure();
     aap.SetAtm(0.53);
     liquid := new Water();
     liquid.SetAap(aap);
     liquid.SetBoilingPoint();
     liquid.SetTemperature(35.0); liquid.SetAmount(1000.0));
end RealWorld
```

System Assets

```
-- SYSTEM-SW-controller
class Software
instance variables
  heater : Heater;
  thermistor : Thermistor;
  liquid_level_sensor : LiquidLevelSensor;
operations
  public Setup: RealWorld ==> ()
  Setup(realworld) ==
    (heater := new Heater();
     heater.Setup(realworld);
     thermistor := new Thermistor();
     thermistor.Setup(realworld);
     liquid_level_sensor := new LiquidLevelSensor();
     liquid_level_sensor.Setup(realworld) );
  public Boil: () ==> ()
  Boil() ==
    while thermistor.GetTemperature() < 100.0 and
          liquid_level_sensor.IsOn() = true
      do heater.On()
  pre  liquid_level_sensor.IsOn() = true
  post liquid_level_sensor.IsOn() = true;
end Software

-- SYSTEM-HW-heater
class Heater
types
  Switch = <On> | <Off>;
instance variables
  sw : Switch;
  realworld_liquid : Liquid;
operations
  public Setup: RealWorld ==> ()
  Setup(realworld) ==
    realworld_liquid := realworld.liquid;
  public On: () ==> ()
  On() ==
   (sw := <On>;
    realworld_liquid.AddTemperature());
  public Off: () ==> ()
  Off() ==
    sw := <Off>;
end Heater

-- SYSTEM-HW-thermistor
class Thermistor
instance variables
  realworld_liquid : Liquid;
operations
  public Setup: RealWorld ==> ()
  Setup(realworld) ==
    realworld_liquid := realworld.liquid;
  public GetTemperature: () ==> real
  GetTemperature() ==
    return realworld_liquid.GetTemperature();
end Thermistor

-- SYSTEM-HW-liquid-level-sensor
class LiquidLevelSensor
instance variables
  realworld_liquid : Liquid;
operations
  public Setup: RealWorld ==> ()
  Setup(realworld) ==
    realworld_liquid := realworld.liquid;
  public IsOn: () ==> bool
  IsOn() ==
    return realworld_liquid.GetAmount() > 0;
end LiquidLevelSensor
```

Context assets

```
-- CONTEXT-atmospheric-air-pressureplace
class AtmosphericAirPressure
instance variables
  protected atm : real;
operations
  public GetAtm: () ==> real
  GetAtm() == return atm;
  public SetAtm: real ==> ()
  SetAtm(a) == atm := a;
end AtmosphericAirPressure

-- CONTEXT-atmospheric-air-pressureplace-normal
class NormalAtmosphericAirPressure
  is subclass of AtmosphericAirPressure
instance variables inv atm = 1
end NormalAtmosphericAirPressure

-- CONTEXT-atmospheric-air-pressureplace-high
class HighAtmosphericAirPressure
  is subclass of AtmosphericAirPressure
instance variables inv atm > 1
end HighAtmosphericAirPressure

-- CONTEXT-atmospheric-air-pressureplace-low
class LowAtmosphericAirPressure
  is subclass of AtmosphericAirPressure
instance variables inv atm < 1
end LowAtmosphericAirPressure

-- CONTEXT-liquid
class Liquid
instance variables
  protected aap : AtmosphericAirPressure;
  protected boiling_point : map real to real;
  protected temperature : real;
  protected amount : real;
operations
  public GetAap: () ==> AtmosphericAirPressure
  GetAap() == return aap;
  public SetAap: AtmosphericAirPressure ==> ()
  SetAap(a) == aap := a;
  public GetBoilingPoint: real ==> real
  GetBoilingPoint(atm) == return boiling_point(atm);
  public GetTemperature: () ==> real
  GetTemperature() == return temperature;
  public SetTemperature: real ==> ()
  SetTemperature(t) == temperature := t;
  public AddTemperature: () ==> ()
  AddTemperature() ==
    if temperature < boiling_point(aap.GetAtm())
      then  temperature := temperature + 1.0
      else (temperature := boiling_point(aap.GetAtm());
            amount := amount - 1.0  --- evaporation
           )
  pre  temperature <= boiling_point(aap.GetAtm())
  post temperature <= boiling_point(aap.GetAtm());
  public GetAmount: () ==> real
  GetAmount() == return amount;
  public SetAmount: real ==> ()
  SetAmount(a) == amount := a;
end Liquid

-- CONTEXT-liquid-water
class Water is subclass of Liquid
operations
  public SetBoilingPoint: () ==> ()
  SetBoilingPoint() ==
    boiling_point := {1.0  |-> 100.0, 0.53 |-> 85.0};
end Water
```

Configuring Software Product Line Feature Models Based on Stakeholders' Soft and Hard Requirements

Ebrahim Bagheri, Tommaso Di Noia, Azzurra Ragone, and Dragan Gasevic

NRC-IIT, Politecnico di Bari, University of Trento, Athabasca University

Abstract. Feature modeling is a technique for capturing commonality and variability. Feature models symbolize a representation of the possible application configuration space, and can be customized based on specific domain requirements and stakeholder goals. Most feature model configuration processes neglect the need to have a holistic approach towards the integration and satisfaction of the stakeholder's soft and hard constraints, and the application-domain integrity constraints. In this paper, we will show how the structure and constraints of a feature model can be modeled uniformly through Propositional Logic extended with concrete domains, called $\mathcal{P}(\mathcal{N})$. Furthermore, we formalize the representation of soft constraints in fuzzy $\mathcal{P}(\mathcal{N})$ and explain how semi-automated feature model configuration is performed. The model configuration derivation process that we propose respects the soundness and completeness properties.

1 Introduction

Software product line engineering (SPLE) is concerned with capturing the commonalities, universal and shared attributes of a set of applications for a specific domain [1]. It allows for the rapid development of variants of a domain specific application through various configurations of a common set of reusable assets often known as *core assets*. In SPLE, *feature modeling* is an important technique for modeling the attributes of a family of systems [2]. It provides for addressing commonality and variability both formally and graphically, allows for the description of interdependencies of the product family attributes (features) and the expression of the permissible variants and configurations of the product family. By reusing domain assets as a part of the feature model configuration process, a new product can be developed in a shorter time at a lower cost. Large-scale industrial applications of software product families entail the development of very large feature models that need to be customized before they can be used for a specific application. In order to develop an instance of a product family from the relevant feature model, its most desirable features need to be selected from the feasible configuration space of the product family. The selection of the best set of features for a product would be based on the strategic goals, requirements and limitations of the stakeholders, as well as the integrity constraints of the feature model. Once the desired features are specified, the feature model can be customized such that it includes the wanted features and excludes the non-relevant ones. A final fully-specific feature model with no points for further customization is called a *configuration*. In many cases, a configuration is gradually developed in several stages. In each stage, a subset of the preferred features are selected and finalized and the unnecessary features are discarded.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 16–31, 2010.

This feature model is referred to as a *specialization* of the former feature model and the staged refinement process constitutes *staged configuration* [2]. Despite the effectiveness of staged configuration, it is still hard to manually create configurations for industrial-scale feature models. The reason is multifold:

1. In a large feature model, it is infeasible for a group of experts to keep track of all the mutual interdependencies of the features; therefore, understanding the implications of a feature selection decision becomes very difficult. In other words, selecting the feature that would maximize the satisfaction of the stakeholders and at the same time minimize the unintended consequences is both important and complicated;
2. Understanding the requirements and needs of the stakeholders and attempting to satisfy them simultaneously can be viewed as a complex constraint satisfaction problem. In cases where the stakeholders have multiple requests, ensuring that all of the requests have been satisfied is a complex task;
3. Consistency checking and verification of a given configuration for a feature model is very time consuming (with high computational complexity) and error prone on large-scale feature models. This is due to the need for performing many cross-reference integrity and stakeholder constraint checks on the developed configuration. The configuration needs to be checked against the feature model constraints to see whether it respects their enforcement in terms of the inclusion of all mandatory features and the exclusion of undesired features, and should also be verified with regards to the stakeholders stated requirements and restrictions.

Here, we attempt to show how a semi-automated approach to feature model configuration (*interactive configuration*), based on a fuzzy propositional language $\mathcal{P}(\mathcal{N})$ is able to address the aforementioned concerns. This paper attempts to create an interactive feature model configuration process. Most of the interactive configuration procedures in the literature mainly focus on satisfying and analyzing the stakeholders hard constraints and also validating the consistency of a developed feature model configuration. Therefore, decision making in tradeoff situations where the choice between competing features needs to be made cannot be formally supported by these approaches. As we will discuss, in cases where a choice needs to be made between several competing features, stakeholders' soft constraints can help in making the right decision. This is the fact that has been neglected in almost all other available approaches in the literature. It is important to take a holistic approach towards the satisfaction of the stakeholder's soft and hard constraints, and the application-domain integrity constraints, which would assist the modelers in making the best feature selection decisions. As the main contributions we show how:

1. $\mathcal{P}(\mathcal{N})$ is a suitable language to express the structure and the integrity constraints of a feature model, as well as the hard constraints of the stakeholders;
2. The soft constraints (preferences) of the stakeholders are represented using a fuzzy extension of $\mathcal{P}(\mathcal{N})$, which would allow for a more relaxed reasoning procedure;
3. Formal and fuzzy reasoning techniques are employed to develop a sound and complete interactive feature model configuration process.

The spotlight of this paper is that it is a novel work which considers the stakeholders' soft constraints, i.e. desired quality attributes, while configuring a feature model. It uses

a variant of propositional logics along with fuzzy logics to represent feature models and their quality attributes and to be able to bring the stakeholders' soft and hard constraints under one umbrella. Capturing both soft and hard constraints allows our proposed approach to be the first of its kind to simultaneously consider integrity constraints, stakeholder requests and quality attributes during the feature model configuration process, a process which has been formally shown to be *sound* and *complete*.

2 Feature Modeling

Features are important distinguishing aspects, qualities, or characteristics of a family of systems [3]. To form a product family, all the various features of a set of similar/related systems are composed into a feature model. A feature model is a means for representing the possible configuration space of all the products of a system product family in terms of its features. Graphical feature models are in the form of a tree whose root node represents a domain concept, and the other nodes and leafs illustrate the features. In a feature model, features are hierarchically organized and can be typically classified as: *Mandatory*; *Optional*; *Alternative feature group*; *Or feature group*. This tree structure falls short at fully representing the complete set of mutual interdependencies of features; therefore, additional constraints are often added to feature models and are referred to as *Integrity Constraints (IC)*. The two most widely used integrity constraints are: *Includes*: the presence of a given feature requires the *existence* of another feature; *Excludes*: the presence of a given feature requires the *elimination* of another feature. Lopez-Herrejon

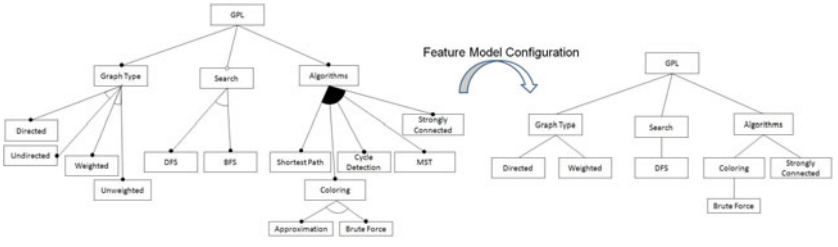


Fig. 1. The graph product line feature model

and Batory have proposed the Graph Product Line (GPL) as a standard problem for evaluating product line methodologies [4]. The intention is to develop configurations of GPL for different problem domains. For instance, GPL can be configured to perform several search algorithms over a directed or undirected graph structure. The graphical representation of GPL is shown in Figure 1. Clearly, not all possible configurations of GPL produce valid graph programs. For instance, a configuration of GPL that checks if a graph is strongly connected cannot be implemented on an undirected graph structure. Such restrictions are expressed in the form of integrity constraints. Some examples of these constraints are: Cycle Detection EXCLUDES BFS; Cycle Detection INCLUDES DFS; Strongly Connected INCLUDES DFS; Strongly Connected INCLUDES Directed; Strongly Connected EXCLUDES Undirected.

3 Formalism for Feature Modeling

There have been different approaches towards the formalization of feature models among which the use of pure propositional logic [5], description logic [6], and iterative tree grammars [7] are the most important ones.

Here we use Propositional Logic enriched with concrete domains as originally proposed in [8]. Interested readers can refer to [8] for more details.

Definition 1 (The language $\mathcal{P}(\mathcal{N})$). *Let $\mathcal{A}$ be a set of propositional atoms, and $\mathcal{F}$ a set of pairs $\langle f, D_f \rangle$ each made of a feature name and an associated concrete domain D_f, and let k be a value in D_f. Then the following formulas are in $\mathcal{P}(\mathcal{N})$:*

1. *every atom $A \in \mathcal{A}$ is a formula in $\mathcal{P}(\mathcal{N})$*
2. *if $\langle f, D_f \rangle \in \mathcal{F}$, $k \in D_f$, and $c \in \{\geq, \leq, >, <, =, \neq\}$ then (fck) is a formula in $\mathcal{P}(\mathcal{N})$*
3. *if ψ and φ are formulas in $\mathcal{P}(\mathcal{N})$ then $\neg\psi$, $\psi \wedge \varphi$ are formulas in $\mathcal{P}(\mathcal{N})$. We also use $\psi \vee \varphi$ as an abbreviation for $\neg(\neg\psi \wedge \neg\varphi)$, $\psi \rightarrow \varphi$ as an abbreviation for $\neg\psi \vee \varphi$, and $\psi \leftrightarrow \varphi$ as an abbreviation for $(\psi \rightarrow \varphi) \wedge (\varphi \rightarrow \psi)$.*

We call $\mathcal{L}_{\mathcal{A},\mathcal{F}}$ the set of formulas in $\mathcal{P}(\mathcal{N})$ built using $\mathcal{A}$ and $\mathcal{F}$. Moreover we call facts, all those singleton formulas containing only a single atom $A \in \mathcal{A}$ or a restriction over a concrete feature (fck).

In order to define a formal semantics of $\mathcal{P}(\mathcal{N})$ formulas, we consider interpretation functions $\mathcal{I}$ that map propositional atoms into $\{\mathsf{true}, \mathsf{false}\}$, feature names into values in their domain, and assign propositional values to numerical constraints and composite formulas according to the intended semantics.

Using $\mathcal{P}(\mathcal{N})$ we can easily represent IS-A and equivalence relations using pure Propositional Logic or involving concrete features. For example,

$$GraphType \wedge Algorithms \leftrightarrow GPL$$

is a $\mathcal{P}(\mathcal{N})$ formula that states that a graph product line configuration is equivalent to the configuration of both a graph type and an algorithm feature.

As it can be seen in $\mathcal{P}(\mathcal{N})$, rules are either satisfied and are true or are false otherwise. In order to be able to represent varying degrees of truthfulness over concrete features, a fuzzy extension of $\mathcal{P}(\mathcal{N})$ can be formulated.

Definition 2 (Fuzzy $\mathcal{P}(\mathcal{N})$). *The alphabet of Fuzzy $\mathcal{P}(\mathcal{N})$ is a tuple $\langle \mathcal{A}, \mathcal{F}, \mathcal{C}, \overline{\mu} \rangle$ where:*

- *$\mathcal{A} = \{A_i\}$ and $\mathcal{F} = \{f_j\}$ are defined as for $\mathcal{P}(\mathcal{N})$;*
- *$\mathcal{C} = \{cn_h\}$ is a set of attributes such that $\mathcal{F} \cap \mathcal{C} = \emptyset$;*
- *$\overline{\mu} = \{\mu_{cn_h}^{A_i}\}$ is a set of fuzzy membership functions.*

The following formulas are in fuzzy $\mathcal{P}(\mathcal{N})$:

1. *$cn_h = \mu_{cn_h}^{A_i}$;*
2. *$F \rightarrow \bigwedge_h cn_h = \mu_{cn_h}^{F} \wedge \psi$, with $\psi \in \mathcal{P}(\mathcal{N})$ and $F \in \mathcal{A}$;*

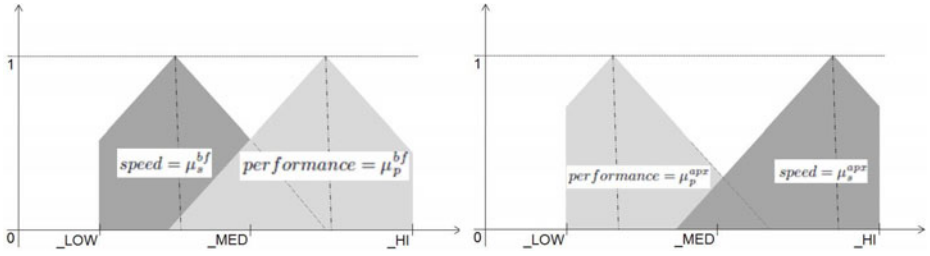


Fig. 2. The Semantic annotation of a) *brute force* and b) *approximation* graph coloring features

If a fuzzy $\mathcal{P}(\mathcal{N})$ formula is in the form 1 we call it fuzzy fact, if it is in the form 2 we call it fuzzy clause.

Some of the more widely used examples of fuzzy membership functions are the triangular, trapezoidal, and Gaussian functions. For each fuzzy predicate $\mu \in \overline{\mu}$ the only restriction is $\mu \in [0, 1]$. For the sake of illustration and due to the simplicity of its representation, we use the triangular function $tri(d_1, d_2, a, b, c)$, where d_1 and d_2 are the domains of the function, a, b, c are the parameters throughout this paper.

In order to clarify the syntax of fuzzy $\mathcal{P}(\mathcal{N})$ formulas we represent the following two fuzzy clauses related to features of the graph product line:

$$Brute\_Force \rightarrow performance = \mu_p^{bf} \wedge speed = \mu_s^{bf} \wedge Coloring \quad (1)$$

$$Approximation \rightarrow performance = \mu_p^{apx} \wedge speed = \mu_s^{apx} \wedge Coloring \quad (2)$$

These two clauses show that *brute force* and *approximation* are two methods for graph coloring. Each of them has been annotated with information about its performance and speed. As mentioned earlier, the information regarding the annotation of the abstract concepts need to be provided by outside sources of information. Figure 2 depicts the fuzzy values of these two features. The membership functions on the left side of the figure basically show that the brute force technique for graph coloring is rather slow but has high performance in terms of accuracy. On the other side, the approximate technique for graph coloring has been described in terms of its higher speed and weaker performance. The ability to annotate abstract models is very important in feature modeling, due to the fact that feature models are abstract representations of a family of products where domain-specific information that would be possible unification options for the open variables of clauses in a first-order format have been removed from the models; therefore, reasoning is only feasible at the abstractions level represented in propositional form and their fuzzy annotations.

4 Conceptual Modeling

The formalization of feature modeling information in our proposed language entails the development of multiple separate knowledge bases. Besides the structural information of the feature model (i.e., feature hierarchies represented as $\mathcal{SKB}$) and integrity constraints between the features ($\mathcal{IC}$), the rest of the knowledge bases are as follows: The

selection of the correct features of a feature model in the configuration process is based on the efficiency of the features to perform the required functional tasks as well as fulfill some of the non-functional requirements which are aligned with the strategic objectives of the stakeholders. In order to be able to understand how each feature relates with the functional and non-functional requirements, we propose to annotate the features. For this reason, we adopt the concept of *concerns* from the Preview framework [9]. Concerns relate with the high-level strategic objectives of the specific application domain and the target product audience; therefore, they can be used to ensure consistency and alignment between the vital goals pursued by the design of a product and the product family features. Simply put, concerns are the desired business quality attributes, which need to be considered through the staged configuration process. Examples of concerns can include to cost, time, risk, volatility, customer importance, etc. For instance, *speed* and *performance* are the two concerns that have been used to annotate the features of GPL in Figure 2. This is because speed and performance are important decision making criteria in the GPL configuration process. Now, since concerns are abstract concepts, the degree of ability of a feature to satisfy a given concern can be expressed in a fuzzy form; therefore, the annotation of features with concerns and their corresponding degrees of satisfaction are shown through fuzzy $\mathcal{P}(\mathcal{N})$ clauses, and the collection of these information is referred to as the *utility knowledge base* ($\mathcal{UKB}$). Utility knowledge depicts how various features of a given feature model relate with and to what extent they are able to satisfy the objectives of the configuration. In our framework, we represent $\mathcal{UKB}$ as a set of fuzzy clauses. Referring back to the GPL example and assuming that the important concerns for product configuration are speed and performance, the fuzzy propositional clauses shown in Equations (1) and (2) are the utility knowledge related to the *Brute Force* and *Approximation* graph coloring features. The information regarding the utility annotation of the feature model should be provided at design time by the domain and/or product family experts by providing statements such as *I believe the brute force graph coloring feature is rather slow* ($speed = \mu_s^{bf}$) *but has an acceptable performance* ($performance = \mu_p^{bf}$) or *I think that although the approximate graph coloring feature has a high execution speed* ($speed = \mu_s^{apx}$)*, it has less accuracy in terms of performance* ($performance = \mu_p^{apx}$). Such statements can be easily represented in fuzzy $\mathcal{P}(\mathcal{N})$, shown in (1) and (2).

Stakeholders and product developers often specify a set of basic features that they want to see in the final product. For instance, in the GPL configuration process, they may require the inclusion of the graph coloring feature. Such requirements are referred to as *hard constraints*. The satisfaction of hard constraints is either feasible or not, which makes the configuration process based on hard constraints a crisp one. However, besides the hard constraints, the stakeholders may also specify their preferences over the defined concerns such as *high speed is very important*, or *lower performance is tolerable*. These kinds of requests are called the *soft constraints* or *preferences*. In this paper, Stakeholders' hard and soft constraints are represented by $\mathcal{SR}_h$, and $\mathcal{SR}_s$, respectively. Similar to utility knowledge, soft constraints can be represented using fuzzy $\mathcal{P}(\mathcal{N})$ facts, e.g., *high speed is very important* can be stated as $\mathcal{SR}_s(speed)$ = tri(_LOW, _HI, _MED, $\frac{\_MED+\_HI}{2}$, _HI), which is a triangular fuzzy membership function whose maximum is located at the $\frac{\_MED+\_HI}{2}$ point; therefore depicting the importance of speed in this case

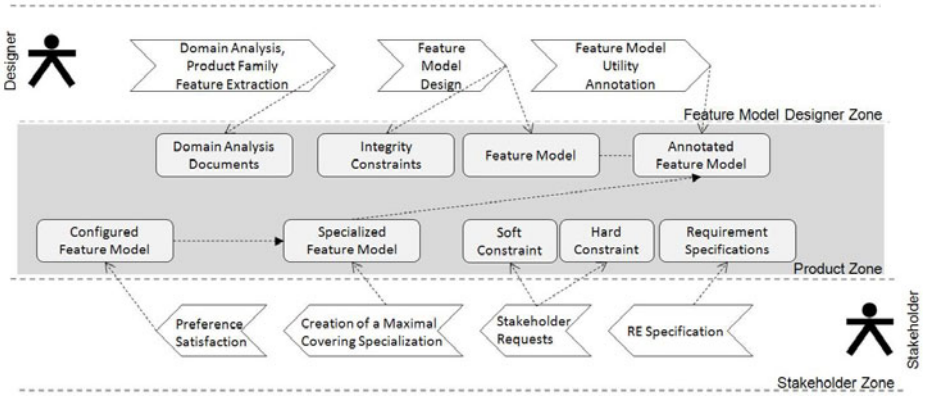


Fig. 3. The overview of the interactive feature model configuration process

(See Figure 2). Summing up w.r.t. to the feature modeling knowledge, we have the following formalization: $\mathcal{SKB}$: the feature model structural knowledge represented using $\mathcal{P}(\mathcal{N})$ axioms; $\mathcal{IC}$: integrity constraints defind through $\mathcal{P}(\mathcal{N})$ formulas; $\mathcal{UKB}$: features utility knowledge as a set of fuzzy $\mathcal{P}(\mathcal{N})$ clauses involving concerns; $\mathcal{SR}_h$: stakeholders' hard constraints as $\mathcal{P}(\mathcal{N})$ facts; $\mathcal{SR}_s$: stakeholders' soft constraints (preferences) as fuzzy $\mathcal{P}(\mathcal{N})$ facts involving concerns.

5 Interactive Feature Model Configuration

The overview of our proposed interactive feature model configuration process is shown in Figure 3. As it can be seen, the feature model designers need to take three steps:

D1. Perform domain analysis to understand the set of all possible features and their interdependencies in the product family members. Available domain analysis methodologies exist that can be used for this purpose;
D2. Design a comprehensive feature model based on the result of the domain analysis that properly supports variability. This would include both the feature model and its accompanying integrity constraints;
D3. Annotate the features with appropriate utility knowledge. Such information would show how each feature can contribute to the satisfaction of the high-level abstract objectives of the problem domain. For instance in GPL and with the speed and performance concerns, the designers would need to show how each feature behaves with respect to these two concerns, e.g., *Finding MST is both fast and accurate*.

Once an annotated feature model is developed, the annotation information can be used to reason about the suitability of a feature for a given purpose. For example, the features that have slow execution speed are not very suitable to be selected for an application that requires realtime performance. The annotation information can go hand in hand with the stakeholders hard and soft constraints to provide the means for an interactive

configuration process. In the context of the interactive configuration process, the stakeholders need to perform the following:

S1. Understand their expectations from the final product within the context of the feature model and clearly specify their requirements, most likely with the help of the model designers or requirement engineers;
S2. Differentiate between their hard constraints, which are vital for the target product, and their soft constraints that can be tolerated if not satisfied;
S3. Develop and analyze the maximal covering specialization based on the stakeholders hard constraints and requests. In view of this specialization, the stakeholders might consider revising their requests to reach a more desirable specialization;
S4. Employ the feature recommendations based on the soft constraints to decide on the set of most appropriate remaining open features to fully configure the feature model. The feature recommendation process guides the stakeholders towards the configuration of the feature model.

In S1 to S4, the interactive configuration procedure benefits from two important steps. In the first important step (S3), the feature model is specialized based on the hard constraints of the stakeholders and a specialization is provided to the stakeholders, which can be useful for them to decide whether they want to change their selected set of hard constraints or not. If the hard constraints are changed, a new specialization is then developed. In the next step (S4), the remaining open features of the specialization (developed in S3) are rank-ordered based on their degree of contribution to the satisfaction of the stakeholders' soft constraints. The features are recommended to the stakeholders according to the rank-order. The stakeholders can interactively select the features they desire until the feature model is fully configured.

Hard Constraints Satisfaction: It is important to satisfy the stakeholders' hard constraints before their soft constraints, since they represent the *required* features of the product. Let us provide ground definitions for the hard constraint satisfaction process.

Definition 3. *Let $c \in \mathcal{SR}_h$ be a hard constraint of the stakeholders, $\mathcal{IC}$, and $\mathcal{SKB}$ be the integrity constraints and structural knowledge of the feature model. The enforcement of c onto $\mathcal{SKB} \cup \mathcal{IC}$, will entail a set of facts called consequential facts of c, denoted $Cons(c)$. We define*

$$Cons(c) = \{c' \mid \mathcal{SKB} \cup \mathcal{IC} \cup \{c\} \models c'\}$$

For example, suppose that the *Cycle Detection* feature is a hard constraint of the stakeholders, which means that the stakeholders want to have this feature in their final product. Based on the structural knowledge of GPL and the integrity constraints we have $Cons(CycleDetection) = \{\neg BFS, DFS, CycleDetection\}$. A crucial implication of the entailed facts of the hard constraints is that the entailed facts of one hard constraint may be inconsistent with the other hard constraints. In other words, given two constraints $c_1, c_2 \in \mathcal{SR}_h$ there is a fact $\overline{c}$ such that both $\overline{c} \in Cons(c_1)$ and $\neg\overline{c} \in Cons(c_2)$. For instance, assume $\mathcal{SR}_h = \{CycleDetection, BFS\}$, then $\mathcal{SR}_h \cup \mathcal{SKB} \cup \mathcal{IC} \models \mathsf{false}$, which means that the consequential facts of this set of hard constraints are inconsistent. As a result, some of the hard constraints expressed by

the stakeholders might be mutually exclusive making the simultaneous satisfaction of all such requests infeasible; therefore, the aim should be to maximize the number of satisfied hard constraints. Formally, given a set of hard constraints $\mathcal{SR}_h$, the idea is to compute a partition of $\mathcal{SR}_h$ such that: 1) $\mathcal{SR}_h = \mathcal{MCS} \cup \mathcal{UT}$; 2) $\mathcal{MCS} \cap \mathcal{UT} = \emptyset$; 3) $\mathcal{MCS} \cup \mathcal{SKB} \cup \mathcal{IC} \not\models \mathsf{false}$; 4) There is no partition $\mathcal{SR}_h = \mathcal{MCS}' \cup \mathcal{UT}'$ such that $\mathcal{MCS} \subset \mathcal{MCS}'$. In order to solve the above problem we should compute all possible assignments $\mathcal{I}_k$ to variables in $\mathcal{A}$ and $\mathcal{F}$ such that $\mathcal{I}_k \models \mathcal{SKB} \cup \mathcal{IC}$ and find the assignment such that the number of variables in $\mathcal{SR}_h$ assigned to *true* is maximal. Algorithm 1 shows the computational procedure. The algorithm calls two functions: COMPUTENEWASSIGNMENT: This function returns a pair $\langle more, \mathcal{I} \rangle$ where $more$ is a Boolean variable which is *true* if there are still assignments to be computed and *false* otherwise and $\mathcal{I}$ is a new assignment for variables in $\mathcal{A}$ and $\mathcal{F}$. SATISFIEDHARDPREFERNCES: Given $\mathcal{SR}_h$ and an assignment $\mathcal{I}$, the function returns the number of variables in $\mathcal{SR}_h$ assigned to *true* with respect to the assignment $\mathcal{I}$.

Algorithm 1. MAXIMALCOVERINGSPECIALIZATION($\mathcal{SR}_h, \mathcal{IC}, \mathcal{SKB}$)

more = *true*
max = 0
$\mathcal{MCS} = \emptyset$
while *more* = *true*
do
- $\langle more, \mathcal{I} \rangle$ = COMPUTENEWASSIGNMENT($\mathcal{SR}_h, \mathcal{IC}, \mathcal{SKB}$)
- **if** $\mathcal{I} \models \mathcal{SKB} \cup \mathcal{IC}$
  - **then if** SATISFIEDHARDPREFERNCES($\mathcal{SR}_h, \mathcal{I}$) > *max*
    - **then**
      - $\mathcal{I}_{max} = \mathcal{I}$
      - *max* = SATISFIEDHARDPREFERNCES($\mathcal{SR}_h, \mathcal{I}$)

for each $A_i \in \mathcal{A}$
do
- **if** $\mathcal{I}_{max} \models A_i$
  - **then** $\mathcal{MCS} = \mathcal{MCS} \cup \{A_i\}$
  - **else** $\mathcal{UT} = \mathcal{UT} \cup \{A_i\}$

return $(\mathcal{I}_{max}, \mathcal{MCS}, \mathcal{UT})$

Algorithm 1 computes all possible feature assignments based on the given feature model structural knowledge, integrity constraints and stakeholders hard constraints, and selects the one that satisfies the most number of stakeholder constraints. The algorithm returns this assignment as the maximal covering specialization ($\mathcal{MCS}$) and the set of unsatisfied hard constraints ($\mathcal{UT}$). Ultimately, $\mathcal{MCS}$ contains the maximal covering specialization of the feature model based on the stakeholders' hard constraints ($\mathcal{SR}_h$), and $\mathcal{UT}$ will contain the set of unsatisfiable hard constraints. Based on ($\mathcal{MCS}$) and ($\mathcal{UT}$), the stakeholders will be able to *interact* with the process by analyzing the resulting specialization of the feature model and deciding whether they would like to change some of their selections. If hard constraints are changed at this stage, the maximal covering specialization will be recalculated accordingly to reflect the new selections. It is possible to see from Algorithm 1 that the process of finding the maximal covering specialization for a given feature model is sound and complete. A specialization process is sound iff the selected features in the final specialization are consistent with the integrity constraints and the structural knowledge of the feature model. It is also

complete iff it will find a specialization that satisfies all of the stakeholders' hard constraints whenever one exists.

Theorem 1 (SOUNDNESS). *The maximum covering specialization $\mathcal{MCS}$ computed by Algorithm 1 is the largest subset of $\mathcal{SR}_h$ such that the set of axioms $\mathcal{MCS} \cup \mathcal{SKB} \cup \mathcal{IC}$ is consistent.*

Proof. The algorithm selects an assignment iff the condition $\mathcal{I} \models \mathcal{SKB} \cup \mathcal{IC}$ is satisfied. Among all these assignments it selects the one maximizing the number of hard constraints, i.e.*, the propositional variables in $\mathcal{SR}_h$, such that their value for the assignment $\mathcal{I}_{max}$ is true.*

Theorem 2 (COMPLETENESS). *If $\mathcal{MCS}$ is the maximum number of hard constraints that can be* true *at the same time, given $\mathcal{SKB}$ and $\mathcal{IC}$ then it is computed by Algorithm 1 Proof. In fact, Algorithm 1 evaluates all of the possible specializations of the feature model given $\mathcal{SBK}$, $\mathcal{IC}$ and $\mathcal{SR}_h$, eliminating the chance for missing a more optimal solution that has not been evaluated by the algorithm.*

Although Algorithm 1 computes the assignment we are looking for, it has a serious computational drawback. We have to **always** compute all possible interpretations (possible feature assignments). This leads to an exponential blow up. Indeed, given $\mathcal{A}$ and $\mathcal{F}$ all possible interpretations are equal to $2^{|\mathcal{A}|} \cdot \prod_{\langle f, D_f \rangle \in \mathcal{F}} |\Delta_c(D)|$ where the first term represents all possible *true/false* assignments to propositional variables in $\mathcal{A}$ while the second term takes into account all possible values to be assigned to concrete features in $\mathcal{F}$. We could be more efficient in the computation of $\mathcal{MCS}$ and $\mathcal{UT}$ if we consider our problem as an instance of MAX-WEIGHTED-SAT [10].

Definition 4 (MAX-WEIGHTED-SAT). *Given a set of atoms $\mathcal{A}$, a propositional formula $\phi \in \mathcal{L}_{\mathcal{A}}$ and a weight function $\omega : \mathcal{A} \longrightarrow \mathbb{N}$, find a truth assignment satisfying ϕ such that the sum of the weights of true variables is maximum.*

We call MAXIMALCOVERINGSPECIALIZATION the instance of MAX-WEIGHTED-SAT:

Definition 5 (MAXIMALCOVERINGSPECIALIZATION). *Given a set of hard constraints $\mathcal{SR}_h$, a structural knowledge base $\mathcal{SKB}$ and a set of integrity constraints $\mathcal{IC}$ find a truth assignment $\mathcal{I} \models \mathcal{SKB} \cup \mathcal{IC}$ such that the number of variables $A \in \mathcal{SR}_h$ with $A^{\mathcal{I}} =$ true is maximum.*

In order to find the maximum number of satisfiable hard constraints in $\mathcal{SR}_h$ we transform our MAX-WEIGHTED-SAT problem into a corresponding Integer Linear Programming (ILP) problem, using the standard transformation of clauses into linear inequalities [11]. For the sake of clarity, here we present the procedure for propositional clauses, as introduced in [11]. In our ILP problem the function to maximize is the one referring to Stakeholder Requests $\mathcal{SR}_h$, as we want to maximize such preferences as much as possible. Hence, the function to be maximized is $sr_h = \sum_i x_i$ where x_i is the corresponding binary variable for each $A_i \in \mathcal{SR}_h$. Therefore, given a solution of the optimization problem, if $x_i = 1$ this means that $A_i = true$, while if $x_i = 0$ then $A_i = false$.

The constraints of the optimization problems are the ones coming both from the structural knowledge base ($\mathcal{SKB}$) and the integrity constraints base ($\mathcal{IC}$). In order to have a set of constraints for our ILP problem we have to encode clauses into linear inequalities, mapping each c_i occurring in a clause ϕ with a binary variable x_i. If c_i occurs negated in ϕ then $\neg c_i$ is substituted with $(1 - x_i)$, otherwise we will need to substitute c_i with x_i. After this rewriting it is easy to see that, considering $\vee$—logical *or*—as classical addition (and $\wedge$ as multiplication), in order to have a clause true the evaluation of the corresponding expression must be a value grater or equal to 1.

With this formalization, it is now easy to create a feature model specialization process based on a given set of stakeholder hard constraints, feature model structural knowledge and integrity constraints by solving this straightforward Integer Linear Problem. As outlined earlier, the stakeholders can go through a repetitive process of changing their expressed constraints and re-generating the feature model specialization until they are satisfied with the outcome. The ILP-based specialization algorithm is now computationally efficient as well as sound and complete [11, p.314].

Soft Constraints Satisfaction: In most cases, the maximal covering specialization of a feature model is not a complete model configuration, i.e., many unbound features remain in the specialization that need to be considered. The suitability of such features needs to be evaluated within the context of the stakeholders' soft constraints. For instance, if the stakeholders are looking for a fast algorithm, and two features are available with similar corresponding functionality, but one is fast and inaccurate and the other is slow but accurate, the former is selected. For this purpose, we have two pieces of valuable information at our disposal, namely utility knowledge of the feature model ($\mathcal{UKB}$), and stakeholders' soft constraints ($\mathcal{SR}_s$). A feature would be more relevant to the stakeholders' objectives if its utility knowledge closely matches those requested in the stakeholders' soft constraints. For instance, assuming that the soft constraint of the stakeholders is to create a cheap software tool, and if we have two functionally-similar features that can be selected, the feature whose utility knowledge shows that its implementation is not costly will be the preferred feature. Also note that, by definition, $\mathcal{UBK} \cup \mathcal{IC} \cup \{mcs\}$ is never inconsistent. There always exists an interpretation $\mathcal{I}$ such that both $\mathcal{I} \models mcs$ and $\mathcal{I} \models \mathcal{UBK} \cup \mathcal{IC}$.

Definition 6. *Let $f \in \mathcal{SKB}$, $f \notin \mathcal{MCS}$ be a feature of the feature model not selected in the maximal covering specialization of the model and $\mathcal{UKB}_f : f \rightarrow \bigwedge_h cn_h = \mu^f_{cn_h} \wedge \psi$ the related fuzzy clause in $\mathcal{UKB}$ such that $\mathcal{UBK} \cup \mathcal{IC} \cup \mathcal{MCS} \models \psi$. We denote with $\mathcal{UKB}^{cn}_f$ the utility annotation of feature f with respect to concern cn, and $\mathcal{SR}^{cn}_s$ the soft constraint of the stakeholders with regards to concern cn. The degree of fitness of f in the context of concern cn, denoted $\mathbb{FIT}(f, cn)$, where $\otimes$ is a fuzzy T-norm operator defined as follows:*

$$\mathbb{FIT}(f, cn) = \mathcal{SR}^{cn}_s \otimes \mathcal{UKB}^{cn}_f.$$

In other words, given the interpretations of $\mathcal{I}^{mcs} \models \mathcal{UBK} \cup \mathcal{IC} \cup \mathcal{MCS}$, we consider all those fuzzy clauses such that for their deterministic part ψ the relation $\mathcal{I}^{mcs} \models \psi$ holds too. Since each feature is annotated with multiple concerns, we can interpret the information through Mamdani-type fuzzy inference [12] to calculate the fitness of a feature over all concerns (cn_i), denoted as $\mathbb{FIT}(f)$:

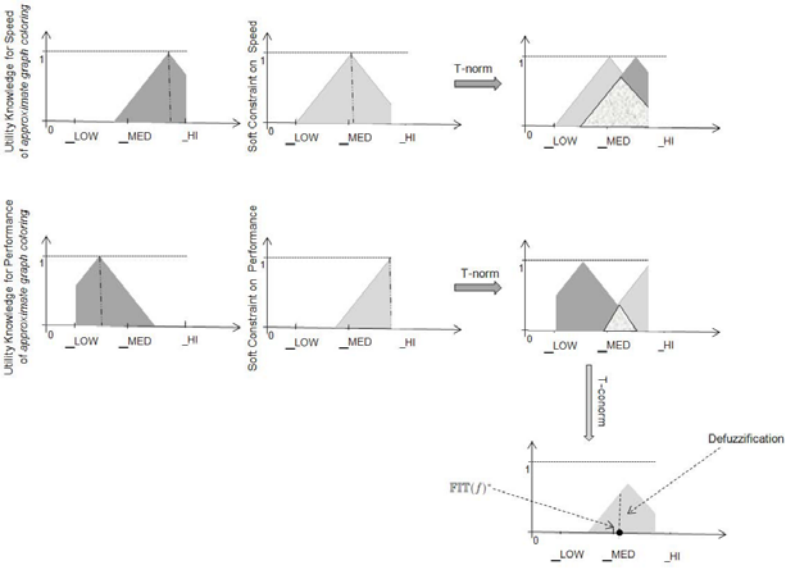


Fig. 4. Computing the fitness of a feature f in context of two concerns: speed and performance

$$\mathbb{FIT}(f) = \bigoplus_{cn_i} \mathcal{SR}_s{}^{cn_i} \otimes \mathcal{UKB}_f{}^{cn_i}.$$

where $\oplus$ is a t-conorm operator. The reason for choosing Mamdani-type fuzzy inference can be explained by its fuzzy representation of both the antecedent and consequence of the rules in contrast with other fuzzy inference methods such as the TSK method [13]. The developed fuzzy fitness measure for each feature can serve as an ordering mechanism for feature prioritization. Priorities can be established by defuzzifying the fuzzy fitness measures through some defuzzifier such as the centroid or maximum defuzzifiers [13]. The corresponding priority value for a feature would be represented by $\mathbb{FIT}(f)^*$. The process of calculating $\mathbb{FIT}(f)^*$ for the approximate feature is depicted in Figure 4 (feature utility knowledge on the left, stakeholder soft constraints in middle, and reasoning results on right). As it can be seen in this figure, the utility knowledge of this feature and the soft constraints of the stakeholders over the two concerns are used for performing the inference. After the Mamdani-type fuzzy inference process, the fitness of the approximate feature for that specific stakeholder is shown to be *Medium*.

Definition 7. *Let f_1, f_2 be two features such that $f_1, f_2 \in \mathcal{SKB}, f_1, f_2 \notin \mathcal{MCS}$. We define $f_1 <_{\mathbb{FIT}} f_2$ as $\mathbb{FIT}(f_1)^* < \mathbb{FIT}(f_2)^*$.*

With the above ordering, it is now possible to extend the maximal covering specialization algorithm to support soft constraints. Algorithm 2 shows the structure of this process called the MAXIMALCOVERINGCONFIGURATION. Algorithm 2 builds on MAXIMALCOVERINGSPECIALIZATION by supporting the satisfaction of soft constraints and providing means for interactive configuration. The features that are not present in the maximal covering specialization are rank-ordered based on their fitness with respect to

the soft constraints of the stakeholders and are recommended to the stakeholders in that order. They can be added to the feature model specialization if and only if they are not conflicting with the features in $\mathcal{MCS}$ computed via MAXIMALCOVERINGSPECIALIZATION.

Algorithm 2. MAXIMALCOVERINGCONFIGURATION($\mathcal{SR}_s, \mathcal{SR}_h, \mathcal{IC}, \mathcal{SKB}, \mathcal{UKB}$)

$\mathcal{MCS} \leftarrow$ MAXIMALCOVERINGSPECIALIZATION($\mathcal{SR}_h, \mathcal{IC}, \mathcal{SKB}$)
$Temp \leftarrow \emptyset$
for each $(A_i \rightarrow \bigwedge_h cn_h = \mu_{cn_h}^{A_i} \wedge \psi \in \mathcal{UKB}$ such that both $\mathcal{IC} \cup \mathcal{SKB} \cup \mathcal{MCS} \models \psi$ and $\mathcal{IC} \cup \mathcal{SKB} \cup \mathcal{MCS} \cup \{A_1\} \not\models \bot)$
do $\begin{cases} Feat \leftarrow \langle A_1, \text{COMPUTEFITNESS}(A_1, \mathcal{SR}_s, \mathcal{UKB})\rangle \\ Temp \leftarrow Temp \cup Feat \end{cases}$
$Feat \leftarrow$ ORDERDESC($Feat, <_{\text{FIT}}$)
$\mathcal{MCC} \leftarrow \mathcal{MCS}$
for $i \leftarrow 0$ **to** SIZE($Feat$)
do $\begin{cases} \langle A, fit\rangle \leftarrow Feat[i] \\ \textbf{if } (\mathcal{IC} \cup \mathcal{SKB} \cup \{A\} \cup \mathcal{MCC} \not\models \bot) \& \text{(Stakeholders Approval)} \\ \quad \textbf{then} \\ \mathcal{MCC} \leftarrow A \cup \mathcal{MCC} \end{cases}$
return $(\mathcal{MCC})$

The stakeholders are able to interact with this algorithm by accepting/rejecting the recommended features. This process is repeated until all features are processed. The algorithm will end by providing a complete feature model *configuration* ($\mathcal{MCC}$).

In summary, the *interactive configuration process* consists of the following steps: 1) Hard and soft constraints of the stakeholders are defined; 2) A maximal covering specialization based on the structural knowledge, integrity constraints and hard constraints of the stakeholders is developed (by MAXIMALCOVERINGSPECIALIZATION); 3) Stakeholders analyze the suitability of the provided specialization. In light of the provided specialization, they can change some of their initial hard constraints in order to gain a more suitable specialization in case of which a new specialization is developed based on the new hard constraints; 4) The remaining unbound features of the feature model are ranked based on the stakeholders' soft constraints and are recommended to the stakeholders (using MAXIMALCOVERINGCONFIGURATION).

5.1 An Illustrative Example

Lets suppose that a group of stakeholders are interested in creating a software graph manipulation package, which is able perform graph coloring, and breadth-first search and also checking the strongly-connectedness property of a weighted graph. Further, suppose that they are able to compromise speed for performance. We would need to elicit the hard and soft constraints of the stakeholders along with the utility knowledge of the features of GPL. As will be seen, we only need the utility knowledge of the two child features of the *graph coloring* feature in this example, namely $ColoringApproximation$ and $BruteForceColoring$; therefore, the information given in Figure 2 will be used. The stakeholder hard constraints can be represented as:

$$\mathcal{SR}_h = \{GraphColoring, StronglyConnected, Weighted, BFS\},$$

which means that these four mentioned features are strictly required by the stakeholders to be included in the final product configuration. Since it has been expressed that the stakeholders prefer performance to speed, it can be inferred that performance has a higher priority compared to speed. This soft constraints can be shown as

$$\mathcal{SR}_s = \{performance = \mu^p_{\mathcal{SR}_s}, speed = \mu^s_{\mathcal{SR}_s}\},$$

where

$$\mu^p_{\mathcal{SR}_s} = tri(\_LOW, \_HI, \_MED, \frac{\_MED + \_HI}{2}, \_HI),$$
$$\mu^s_{\mathcal{SR}_s} = tri(\_LOW, \_HI, \_LOW, \_LOW, \_MED).$$

Given these information, we are now able to compute the maximal covering configuration of GPL: In the first step, the stakeholders' hard constraints, the integrity constraints and the structural knowledge of the feature model are automatically converted into an integer linear program, and the problem of finding the maximal covering specialization is turned into finding a variable assignment that finds an assignment that satisfies the maximum number of stakeholders requests. Such an assignment is added to $\mathcal{MCS}$, and the unsatisfiable facts are added to $\mathcal{UT}$. The result is:

$$\mathcal{MCS} = \{Weighted, GraphColoring, DFS, \neg BFS, StronglyConnected, Directed, \neg Undirected\}$$
$$\mathcal{UT} = \{BFS\}$$

which shows that all hard constraints other than BFS have been satisfied in the developed specialization of GPL. The stakeholders are now able to view and analyze the developed specialization and decide whether they want to continue with it or desire to change the hard constraints to gain BFS in trade for some other features.

In the second step, assuming that the specialization is accepted, the remaining features are ranked based on $<_{\mathbb{FIT}}$ and recommended to the stakeholders. The open features that need to be considered are $ColoringApproximation$ and $BruteForceColoring$. In light of the utility knowledge ($\mathcal{UKB}$) provided by the stakeholders, we can infer that $BruteForceColoring <_{\mathbb{FIT}} ColoringApproximation$; therefore, with priority given to $BruteForceColoring$, it will be recommended to the stakeholders first, and if not selected, $ColoringApproximation$ is then suggested. The selection of any of these features will complete the configuration of the GPL feature model based on the hard and soft constraints of the stakeholders. The right-side of Figure 1 depicts the final configuration of GPL after the selection of the $BruteForceColoring$ feature.

6 Related Work

Feature model configurations can be verified using Logic Truth Maintenance Systems (LTMS) in their representation in the form of propositional formula [14,15,16]. Three of the most widely used methods in this area are Constraint Satisfaction Problem (CSP) solvers [17], propositional SATisfiability problem (SAT) solvers [18], and the Binary Decision Diagrams (BDD) [5]. The basic idea in CSP solvers is to find states (value assignments for variables) where all constraints are satisfied. Although being the most

flexible proposal for verifying feature model configurations, they fall short in terms of performance time on medium and large size feature models [19]. Somewhat similar to CSP solvers, SAT solvers attempt to decide whether a given propositional formula is satisfiable or not, that is, a set of logical values can be assigned to its variables in such a way that makes the formula true. SAT solvers are a good option for manipulating feature models since they are becoming more and more efficient despite the NP-completeness of the problem itself [16]. Closely related is the employment of Alloy [20] for analyzing the properties of feature models that is based on satisfiability of first-order logic specifications converted into boolean expressions [21]. Also, BDD is a data structure for representing the possible configuration space of a boolean function, which can be useful for mapping a feature model configuration space. The weakness of BDD is that the data structure size can even grow exponentially in certain cases; however, the low time performance results of BDD solvers usually compensates for their space complexity.

More closely related to the theme of this paper, Czarnecki et al. have developed probabilistic feature models where a set of joint probability distributions over the features provide the possibility for defining hard and soft constraints [22]. The joint probability distributions are mined from existing software products in the form of Conditional Probability Tables (CPT); therefore, such tables reveal the tendency of the features to be seen together in a software product rather than desirability, i.e., two features may have been seen together in many configurations of a feature model in the past, but they are not desirable for the given product description on hand. Hence, probabilistic feature models are ideal for representing configuration likelihood but not desirability, which is the core concept of the proposal of our paper. Our paper focuses on the strategic objectives of the stakeholders denoted as concerns and tries to align the best possible feature matches to those concerns; therefore, it addresses desirability rather than likelihood. The concepts of the current paper is more closely related to weighted feature models introduced in [23].

7 Concluding Remarks

The research community has put much emphasis on developing methods for the syntactical validity checking of model configurations. These methods mainly focus on forming grammatical correspondences for the graphical representation of feature models and perform automated syntactical analysis based on the model integrity constraints. However, considering the strategic objectives and goals of the stakeholders and the specific domain requirements in the feature model configuration process can ensure that the preferences of the target audience of the product are met as well. In this paper, we have introduced and proposed the use of the fuzzy extension of $\mathcal{P}(\mathcal{N})$ in order to capture both hard and soft constraints of the stakeholders. On this basis, we have developed a maximal covering specialization algorithm that creates a specialization of a feature model based on stakeholders hard constraints, which is complemented by the maximal covering configuration algorithm that orders and creates a sound and complete configuration given the constraints of the stakeholders. The focus of the developed techniques is to achieve maximum desirability for the developed feature model configuration for the stakeholders. Hence, stakeholders' objectives take center place in the proposed methods where they are matched against the utility knowledge of the features.

References

1. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
2. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 266–283. Springer, Heidelberg (2004)
3. Kang, K., Lee, J., Donohoe, P.: Feature-oriented product line engineering. IEEE Software 19, 58–65 (2002)
4. Lopez-Herrejon, R., Batory, D.: A standard problem for evaluating product-line methodologies. In: Bosch, J. (ed.) GCSE 2001. LNCS, vol. 2186, pp. 10–24. Springer, Heidelberg (2001)
5. Mendonca, M., Wasowski, A., Czarnecki, K., Cowan, D.: Efficient compilation techniques for large scale feature models. In: International Conference on GPCE, pp. 13–22 (2008)
6. Wang, H., Li, Y., Sun, J., Zhang, H., Pan, J.: Verifying feature models using OWL. Web Semantics: Science, Services and Agents on the World Wide Web 5, 117–129 (2007)
7. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
8. Ragone, A., Noia, T.D., Sciascio, E.D., Donini, F.M.: Logic-based automated multi-issue bilateral negotiation in peer-to-peer e-marketplaces. JAAMAS 16, 249–270 (2008)
9. Sommerville, I., Sawyer, P.: Viewpoints: principles, problems and a practical approach to requirements engineering. Annals of Software Engineering 3, 101–130 (1997)
10. Ausiello, G., Crescenzi, P., Kann, V., Marchetti-Sp, Gambosi, G., Spaccamela, A.M.: Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties (2003)
11. Papadimitriou, C., Steiglitz, K.: Combinatorial Optimization: algorithms and Complexity. Prentice-Hall, Inc., Englewood Cliffs (1982)
12. Mamdani, E.: Application of fuzzy logic to approximate reasoning using linguistic synthesis. In: Sixth International Symposium on Multiple-Valued Logic, pp. 196–202 (1976)
13. Yager, R., Filev, D.: Essentials of fuzzy modeling and control. John Wiley, Chichester (1994)
14. Schobbens, P., Heymans, P., Trigaux, J.: Feature diagrams: A survey and a formal semantics. In: 14th IEEE International Conference Requirements Engineering, pp. 139–148 (2006)
15. Janota, M., Kiniry, J.: Reasoning about feature models in higher-order logic. In: Software Product Line Conference 2007, pp. 13–22 (2007)
16. Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortes, A.: FAMA: Tooling a framework for the automated analysis of feature models. In: VAMOS Workshop, pp. 129–134 (2007)
17. Benavides, D., Trinidad, P., Ruiz-Cortes, A.: Automated reasoning on feature models. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg (2005)
18. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
19. Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortés, A.: A first step towards a framework for the automated analysis of feature models. Technical Report (2006)
20. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. 11, 256–290 (2002)
21. Gheyi, R., Massoni, T., Borba, P.: A theory for feature models in alloy. In: First Alloy Workshop, pp. 71–80 (2006)
22. Czarnecki, K., She, S., Wasowski, A.: Sample spaces and feature models: There and back again. In: SPLC 2008, pp. 22–31. IEEE Computer Society, Washington (2008)
23. Robak, S., Pieczynski, A.: Employing fuzzy logic in feature diagrams to model variability in software product-lines. In: ECBS 2003, pp. 305–311 (2003)

Usage Context as Key Driver for Feature Selection

Kwanwoo Lee[1] and Kyo C. Kang[2]

[1] Department of Information Systems Engineering, Hansung University,
389 Samsun-dong 3ga, Sungbuk-gu, Seoul, 136-792, Korea
kwlee@hansung.ac.kr
[2] Department of Computer Science and Engineering, POSTECH
San 31 Pohang, Kyoungbuk, 790-784, Korea
kck@postech.ac.kr

Abstract. Product derivation in software product line engineering starts with selection of variable features manifested in a feature model. Selection of variable features for a particular product, however, is not made arbitrarily. There are various factors affecting feature selection. We experienced that the usage context of a product is often the primary driver for feature selection. In this paper, we propose a model showing how product usage contexts are related to product features, and present a method for developing such a model during the domain engineering process and utilizing it to derive an optimal product configuration during the application engineering process. An elevator control software example is used to illustrate and validate the concept and the method.

Keywords: feature modeling, product derivation, product usage contexts, commonality and variability.

1 Introduction

The primary goal of feature modeling is to analyze commonalities and variabilities of a software product line in terms of features. Variants (e.g., optional or alternative features) manifested in a feature model (FM) are selected to derive products during application engineering.

Selection of variable features for a particular product is not made arbitrarily. From our experiences of working on several industrial projects, we have observed that product usage contexts often dictate feature selection. For example, flash memory, which is a non-volatile data storage, is primarily used in memory cards and USB drivers. Although storing data in a flash memory is the main functionality, its implementation techniques (e.g., algorithms and data structures) are quite different depending on the usage context of it. In the USB driver's case, fast storage cycle and small block size give it a significant advantage in supporting data integrity, as it can be pulled out any time. In that domain, we experienced that, although the functionality is same, its implementation features depend heavily on the usage context. As another example, elevator products can largely be classified into passenger elevator or freight elevator depending on their main purpose of use. Since passenger elevators have a goal of carrying passengers comfortably, services or devices for comfortable ride are

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 32–46, 2010.

required. On the other hand, freight elevators do not necessarily require comfortable ride. They instead focus on carrying heavy loads safely, which requires special operational functions and devices for handling heavy loads. These examples show that, although the functionality is same, its implementation features are often dictated by the product's usage context, and we need to provide a systematic approach to analyzing usage contexts of a product line and using this information to support product configuration.

There have been several attempts to relating features and contextual factors affecting feature selection. Hartmann, et al. [7] proposed a context variability model which constrains a feature model by describing how contextual variations (e.g., different geographic regions) affect selection of feature variations. Tun et al. [14] used contextual variations as links between requirement variations and feature variations. In summary, both approaches model contextual variations that affect feature selection as a separate contextual FM and relate the contextual FM to the application FM through feature dependencies (e.g., requires and excludes). Although feature dependencies between the contextual FM and the application FM affect selection of features in the application FM, they are not the only one that affects feature selection. Rather than one contextual feature affecting selection of application features, a group of contextual features determines product goals and attributes, which in turn affects selection of application features.

A number of concerns/goals that stakeholders (e.g., customers) have also affect feature selection. Suppose, for example, that diesel and gasoline engines are alternative features of an automobile product line. If fuel efficiency is the only concern a car buyer has, a diesel engine will be selected. However, if the buyer's concern is price, s/he might take a gasoline engine instead. What if the buyer's concern is both fuel efficiency and price? The choice depends on which concern is more important to the buyer.

Originally the FODA method [8] records issues (e.g., concerns or objectives) related to each decision point of a feature model to help users make selections of both optional and alternative features. Similarly, Thurimella, et al. [13] extended the feature model by augmenting selection criteria (e.g., product specific quality concerns), and assessing each variable feature based on its related criteria. Although both approaches used selection criteria to support feature selection decisions, they do not provide a model showing how selection criteria are derived and how they are related to user goals and product features.

This paper starts with the premise that the usage contexts of a product provide useful information to application engineers helping them making decisions during the feature selection process. This information might be manifested as contextual constraints imposed on selectable features or identified as selection criteria (quality attributes). By analyzing various usage contexts of a product, we can systematically extract the information.

This paper is organized as follows. Section 2 defines a model describing how usage contexts are related to product features. Section 3 presents how this model can be constructed during the domain engineering process and utilized to derive an optimal product configuration during the application engineering process. An elevator control software example is used as a running example in section 4. Our approach is compared with other related work in section 5. Finally we conclude this paper with discussion on the presented approach.

2 Usage Context Driven Domain Knowledge Modeling

Usage contexts are treated in this paper as key drivers in selecting variable features for a particular product. In this section, we first explore factors affecting feature selection, and then describe how those factors are related to usage contexts. Based on this understanding, we present a meta-model describing how usage contexts affect feature selection.

2.1 Factors Affecting Feature Selection

A feature model provides configurable options that can be selected for derivation of products in a software product line. Feature selection is the process of determining optimal choices that satisfy product goals and quality requirements. Typically, there are two types of factors that have significant impacts on feature selection: They are quality attributes/customer goals and technical constraints.

Quality attributes may affect selection or rejection of a feature. For example, when an elevator product has quality concerns such as high level of safety and comfort for passengers, various safety devices and speed control algorithms for smooth riding can be selected as product features. On the other hand, if low development cost is the primary concern of the customer, low-cost devices are configured for the product or optional features with high development cost will be excluded.

In addition to quality attributes, technical constraints also affect feature selection by excluding or restricting some variable features. Some of these constraints come from technical capabilities of software and/or hardware features within a product: For example, selection of a certain functional feature requires selection of a certain hardware device feature as the functional feature uses the capabilities provided by the hardware device. Some constraints may come from the outside of a product, i.e., usage contexts, which will be discussed in the following section.

2.2 Usage Context

Informally, usage contexts are any contextual settings in which a product is deployed or used, which can be detailed in terms of user, physical, social, business, operating environments, etc. In this section, we describe how usage contexts are related to quality attributes or constraints that have significant impacts on feature selection.

User contexts may include the target user groups, their profile (e.g., cultural and technical backgrounds), usage patterns, etc. The user context provides important information for determining product quality attributes. For example, if the target users of an elevator are office workers who typically use it when arriving at the building in the morning and leaving the building after work, the elevator may have a set of quality concerns (e.g., minimum waiting time) to address that particular usage pattern (e.g., peak requests for service at rush hour) of the users.

Physical contexts indicate physical environments or locations where a product is deployed and used. For instance, the number of floors is a physical context of an elevator. The capacity and speed of a passenger elevator are related to the floor space and the number of floors of a building. Some constraints on the capacity and speed can be specified considering these physical conditions.

Social contexts include cultural traits and legal constraints. Analyzing various social contexts in which a product is operated helps deciding constraints that affect feature selection. For example, Sabbath prohibition prevents Jews from operating electrical devices when Sabbath is in effect. Thus an elevator used in a Jewish society may have a constraint for providing the feature stopping an elevator automatically at every floor.

Business contexts influence factors, such as price ranges, that affect product design. For example, the elevator market for small apartments may require low-end elevators that use inexpensive devices or do not provide extra features with high implementation costs.

A product may interface with external systems or devices and run on a specific hardware or software platform. A specific operating environment may restrict or require selection of particular functional features.

In summary, by analyzing usage contexts, we can derive not only product quality attributes that should be satisfied by product features but also technical constraints that exclude or restrict selection of product features in a feature model. These product quality attributes and constraints play important roles in selecting a right set of features for a product. The next section discusses how this information can be captured in a domain knowledge model.

2.3 Domain Knowledge Model

As discussed above, the usage context of a product implies required quality attributes or constraints, which, in turn, affect feature selection. Often, this domain knowledge has not been captured during domain engineering. In this section, we describe how this knowledge can be captured as a domain knowledge model.

Fig.1 shows the domain knowledge model describing domain variability in terms of usage contexts, quality attributes, and product features. Usage contexts are related to product features directly through *UC-PF Mapping* or indirectly through a series of mapping relations with *Quality Attribute Feature*.

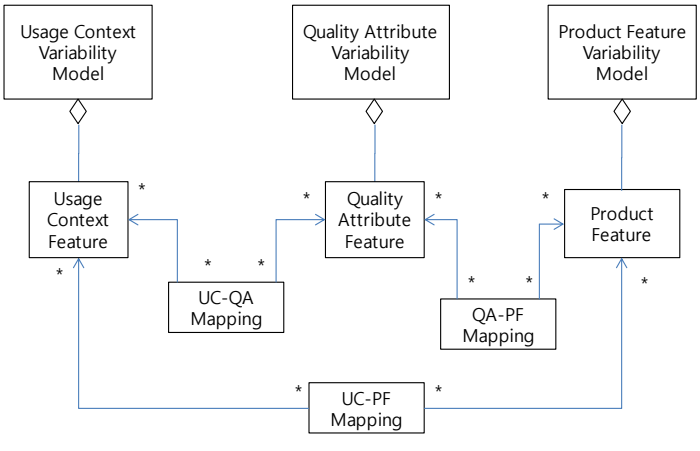


Fig. 1. Domain knowledge model

Before we discuss the modeling elements shown in Fig. 1, we first introduce the feature diagram (FD) used in this paper.

Feature Diagram (FD). This diagram captures commonalities and variabilities of a product line in terms of features, which describe product characteristics from various stakeholder views: end-user, customer, analyst, architect, developer, etc. A feature can have a set of attributes, each of which is defined in terms of a variable name, value range of the variable (the range of possible values that the variable takes), and, optionally, a default value.

Features are organized into a FD using several relationships (i.e., Aggregation, Generalization, and Implemented-by) [9]. In addition, FD specifies allowed feature varibilities through various feature dependencies (Mandatory, Alternative, OR, Requires, Excludes, etc.). Feature selection is the process of determining a valid feature set that satisfies all the feature dependencies specified in a FD. Note that assigning attribute values of a selected feature is also a part of the feature selection process. FD has typically been used to model the problem space or the solution space of a product line. In this paper, we extend the usage of FD to model the usage contexts of the products of a product line.

Usage Context Variability Model (UCVM). This model describes the variability of the contexts in which products of a product line are deployed or used. The usage context variability can be classified into user, physical, social, business, operating contexts, which are described in section 2.2. FD is used to represent a UCVM, in which each type of usage contexts is modeled as a *usage context* (*UC*) *feature*. A detailed example will be provided in section 4.

Quality Attribute Variability Model (QAVM). This model describes the variability of the quality attributes that products of a software product line must satisfy. Each quality attribute is also modeled as a *quality attribute* (*QA*) *feature* with an attribute *weight* that indicates how important the quality attribute is for a specific product. A QA feature can be refined into a set of quality concerns or scenarios, i.e., as sub-QA features. In case all sub-QA features are required to satisfy their parent QA feature, we can model this as "mandatory dependency" between them. Also in case a QA feature can be achieved by only one of its sub-QA features, "alternative dependency" can be specified between the QA feature and its sub-QA features. In this way, the variability of quality attributes is modeled using FD.

Product Feature Model (PFM). This model describes the variability of a product line in terms of *product features*, which can be classified into capabilities, operating environments, domain technologies, and implementation techniques as suggested in FORM [9]. The most common usage of FD is to represent a PFM.

UC-QA Mapping. The links between UCVM and QAVM are defined in *UC-QA Mapping* which describes how usage contexts are related to quality attributes. At first glance, it seems to be sufficient to have a mapping from each UC feature in a UCVM to a QA feature in a QAVM. However, we need to deal with more complex cases. For example, a usage context may drive multiple quality attributes, or a combination of different usage contexts might imply particular quality attributes. In general we can describe n-to-m mapping. Therefore, a mapping between UC features and QA features is defined as follows.

Definition 1. Let F_{UCVM} and F_{QAVM} be the sets of features in *UCVM* and *QAVM* respectively. Further, let C_{UCVM} and C_{QAVM} be the power sets of F_{UCVM} and F_{QAVM}. *UC-QA Mapping*, $M_{UC\text{-}QA}$ is defined as a binary relation on $C_{UCVM} \times C_{QAVM}$. For any *UC* in C_{UCVM} and *QA* in C_{QAVM}, then $(UC, QA) \in M_{UC\text{-}QA}$ indicates that the set of usage contexts *UC* drives the set of quality attributes *QA*.

For example, let *uc1*, *uc2*, and *uc3* be UC features in a UCVM, and *q1* and *q2* be QA features in a QAVM. $(\{uc1\}, \{q1, q2\}) \in M_{UC\text{-}QA}$ represents that *uc1* requires *q1* and *q2*. $(\{uc2, uc3\}, \{q2\}) \in M_{UC\text{-}QA}$ indicates that both *uc2* and *uc3* require q2.

QA-PF Mapping. The links between QAVM and PFVM are defined in *QA-PF Mapping* which describes how quality attributes are related to product features. Some product features may work for or against each QA feature. This leads to the following definition.

Definition 2. *QA-PF Mapping*, $M_{QA\text{-}PF}$ is defined as a ternary relation on $F_{QAVM} \times F_{PFVM} \times R$. F_{QAVM} and F_{PFVM} *are* the set of QA features i*n QAVM* and the set of product features in *PFVM,* respectively, and *R* is a set of impact weight values, i.e., make (++), help (+), hurt(-), and break(--) [4].

For example, let *q1* be a QA feature in a QAVM and *f1* be a product feature in a PFVM. Then $(q1, f1, help) \in M_{PG\text{-}PF}$ indicates that the product feature *f1* has a positive impact for the achievement of the quality attribute *q1*. However, this does not imply that *f1* has to be selected, because it may have negative impacts on achievement of other product quality attributes.

UC-PF Mapping. A usage context directly requires or excludes selection of a product feature or a set of product features. These constraints are mostly n-to-m mappings.

Definition 3. Let C_{UCVM} and C_{PFVM} be the power sets of F_{UCVM} and F_{PFVM}. *UC-PF Mapping*, $M_{UC\text{-}PF}$ is defined as a ternary relation on $C_{UCVM} \times C_{PFVM} \times C_{PFVM}$. Let $UC \in C_{UCVM}$, $RF \in C_{PFVM}$, $EF \in C_{PFVM}$, and $RF \cap EF = \varnothing$, then $(UC, RF, EF) \in M_{UC\text{-}PF}$ indicates that the set of usage contexts *UC* requires selection of all features in *RF* and exclude all features in *EF*.

For example, let *uc1* and *uc2* be optional usage context features in a *UCVM* and ***f1*** and ***f2*** be optional product features in a ***PFVM***. Then $(\{uc1, uc2\}, \{f1\}, \{f2\}) \in M_{UC\text{-}PF}$ means that if the usage context features *uc1* and *uc2* are selected, the product feature *f1* needs to be selected while the product feature *f2* must not be selected.

In the next section, we discuss how these models are constructed.

3 Method Activities

In this section, we provide an overview of the proposed approach in terms of method activities and their related artifacts. Software product line engineering (SPLE) consists of two processes: domain engineering and application engineering. Domain engineering consists of domain analysis, domain design, and domain implementation, while application engineering consists of product analysis (feature configuration), product design, and product implementation. In this paper, we focus on domain and product analysis.

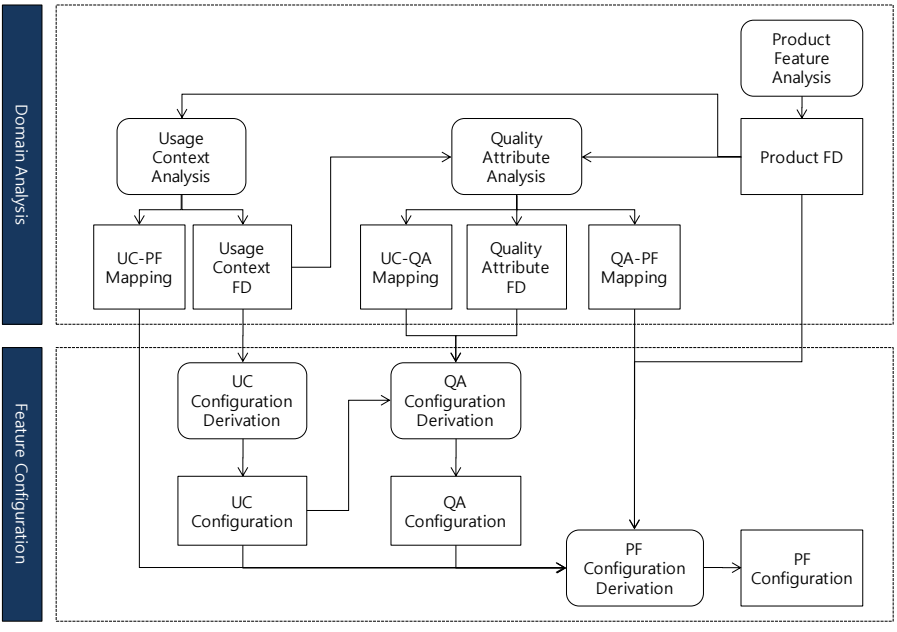


Fig. 2. Method activities and related artifacts

Domain analysis is the process of analyzing software products in a domain to find their common and variable parts. In this paper, commonality and variability of software products are identified and analyzed in terms of usage contexts, quality attributes, and product features.

Domain analysis starts with *Product Feature Analysis*. During *Product Feature Analysis*, a *Product FD* is created, which describes commonalities and variabilities of products in a product line in terms of product features. For each selectable feature of the *Product FD*, key driving factors for feature selection are analyzed during *Usage Context Analysis* and *Quality Attribute Analysis*.

During *Usage Context Analysis*, a *Usage Context FD* is created, which describes contextual variabilities of products of a product line in terms of user, physical, social, business, and operating contexts. In addition, as some usage contexts may constrain selection of product features, these constraints are specified in *UC-PF Mapping*. Note that a *UC Configuration* (a valid set of usage context features) derived from the *Usage Context FD* indicates the usage context of a particular product.

During *Quality Attribute Analysis*, each *UC Configuration* is further analyzed to identify related quality attributes. The output of this activity is a *Quality Attribute FD*, which describes the variability of quality attributes of products in a product line, and *UC-QA Mapping*, which describes mappings between *UC Configurations* and *QA Configurations*. In addition, as a feature may have positive or negative impacts on the achievement of one or more quality attributes, relationships between product features and quality attributes are specified in *QA-PF Mapping* (with impact weights).

With created domain artifacts as input, we can start the feature selection process to derive a *PF Configuration*. The first step (*UC Configuration Derivation*) is to decide

a *UC Configuration* from the *Usage Context FD* by selecting usage context features that are required for a particular product to be derived. After the usage context of the product is configured, *QA Configuration Derivation* can be automatically performed using *UC-QA Mapping*. The final step (*PF Configuration Derivation*) is to include or exclude features specified in *UC-PF Mapping*. This reduces the number of choices that have to be made. In addition, product features are configured by evaluating features in the *Product FD* with consideration of the identified quality features.

Note that during *QA Configuration Derivation* and *PF Configuration Derivation*, conflicting configuration decisions need to be resolved and resolution must be reflected in the domain knowledge model.

The method discussed in this section is illustrated in the next section.

4 Running Example

The example selected in this paper is an elevator control software (ECS) product line. In this section, we present the three variability models and three mappings which we use to describe the domain knowledge of a software product line. Also we discuss how these models are used to derive a product feature configuration.

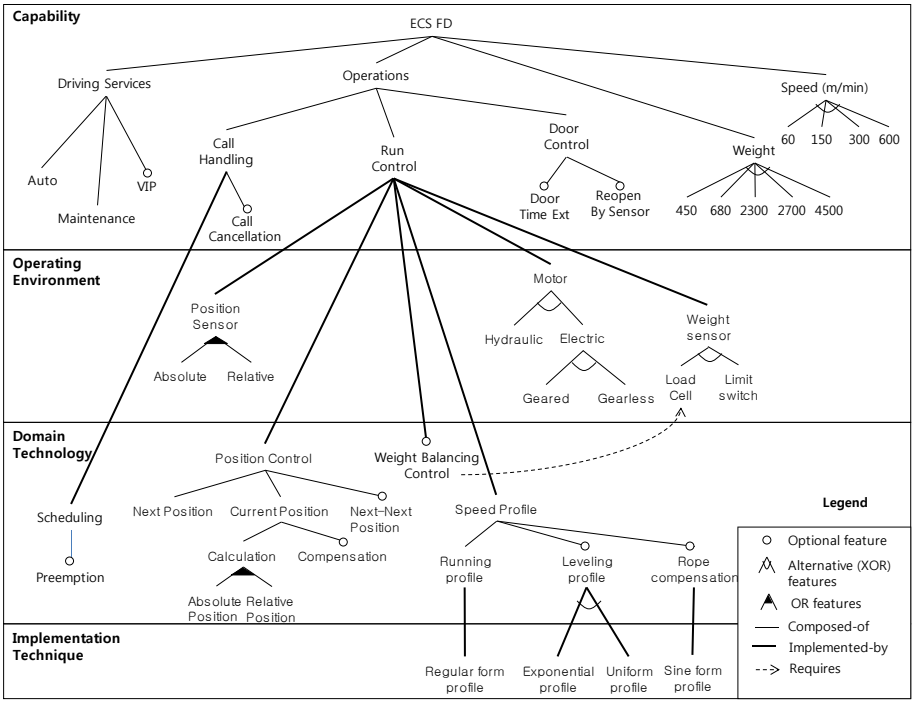


Fig. 3. The product feature diagram of the ECS product line

4.1 Product Feature Analysis

Commonalities and variabilities of a software product line are mainly organized into a *Product FD*. The ECS product line consists of functional features (i.e., *Driving Services* and *Operations*) and non-functional features (i.e., *Weight* and *Speed*), as shown in Fig. 3. A functional feature can be implemented with different sets of operating environment and/or technical features. For example, *Run Control* can be implemented with several operating environment features (e.g., *Position Sensor*) and technical features (e.g., *Position Control*).

4.2 Usage Context Analysis

As described earlier, the contextual variability of a product line can be analyzed in terms of user, physical, social, business, and operating contexts. Fig. 4 shows the *Usage Context FD* that describes the contextual variability of the ECS product line used to illustrate the method. In the example, *Carrying Objects* and *# of Floors* are physical context features. On the other hand, *Market* represents a business context feature. *Carrying Objects* can be *Passenger*, *Freight*, or both. *Passenger* can be further refined into *Office Worker*, *APT Resident*, or *Handicapped*, of which at least one must be selected for a particular product. *# of Floors* can be either *<=10* or *>10*, and *Market* can also be either *Asia* or *Europe*.

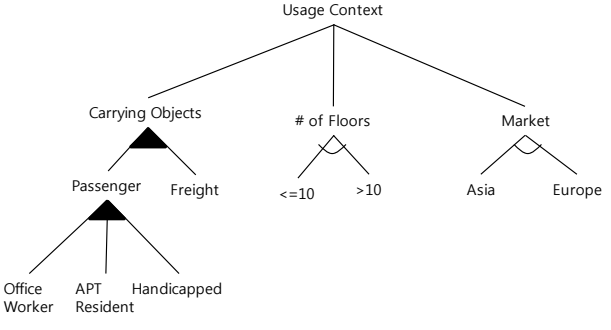


Fig. 4. The usage context feature diagram of an elevator product line

Since usage context features may have influences on selection of product features, we need to analyze what usage context features constrain selection of product features. For example, the last row of Table 1 describes that a set of usage context features (*Asia*, *>10*, *Passenger*, *Office Worker*) requires selection of the feature set (*Electric* and *Gearless*) and excludes selection of the feature set (*60*, *450*, and *4500*), as passenger elevators of office buildings with more than ten floors in the Asian market requires gearless motors, a speed faster than 60, and a weight heavier than 450 but lighter than 4500. (Note here that the features *Speed* and *Weight* may well be modeled as attributes, and *60*, *450*, and *4500* as attribute values. For the purpose of visibility and illustration, we modeled them as features.)

Table 1. The *UC-PF Mapping* of the ECS product line

Usage Contexts	Required Features	Excluded Features
{Asia, Passenger}		{60(Speed)}
{<=10, Passenger}		{300(Speed), 600(Speed), 2300(Weight), 2700(Weight), 4500(Weight)}
{>10, Passenger}		{60(Speed), 450(Weight), 4500(Weight)}
{Freight}		{450(Weight), 680(Weight)}
{Asia, >10, Passenger, Office Worker}	{Electric, Gearless}	{60(Speed), 450(Weight), 4500(Weight)}

4.3 Quality Attribute Analysis

The next step is to identify quality attributes implied by the usage contexts. Table 2 shows a small set of mappings between usage contexts and quality attributes. For example, the usage context *Handicapped* requires quality attributes *Position Accuracy* and *Usability for Handicapped*, as a handicapped person on a wheelchair may want to get in or out of an elevator safely and easily. These mappings must be specified consistently. Consistency of the mappings can be analyzed by translating them into logical expressions [2], which is out of the scope of this paper.

Table 2. The *UC-QA Mapping* of the ECS product line

Usage Contexts drive	Quality Attributes
{Passenger}	{Door Safety, Usability}
{Handicapped}	{Position Accuracy, Usability For Handicapped}
{Office Worker}	{Minimum Waiting Time, Usability for VIPs}
{APT Resident}	{Usability for APT Resident}
{Asia, Passenger,>10}	{Comfort, Position Accuracy}
{Europe}	{Energy Efficiency, Low Cost}
{Freight}	{Position Accuracy}

After identifying quality attributes, their variability needs to be modeled with FD. Fig. 5 shows variabilities of the quality attributes of the ECS product line. In this example, sub-features of *Usability* are modeled as *OR* features (at least one of them must be selected), while others are modeled as optional features.

Some features in the *Product FD* can have positive or negative impacts on some quality attributes specified in the *Quality Attribute FD*. Table 3 shows mappings between some features in the *Product FD* (Fig. 3) and their related quality attribute features in the *Quality Attribute FD* (Fig. 5). For example, *Position Sensor* has two alternative sub-features *Absolute* and *Relative*. Absolute position sensors are better than relative position sensors in terms of accuracy but are more expensive. The decision on which one of them will be selected may vary depending on which quality attribute is more important for a product.

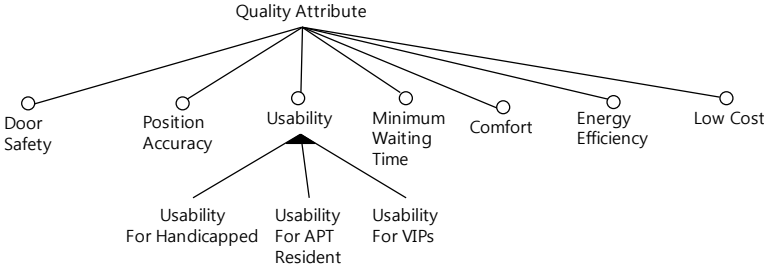


Fig. 5. The quality attribute feature diagram of the ECS product line

Table 3. The *QA-PF Mapping* of the ECS product line

Product Features \ Quality Attributes		Position Accuracy	Door Safety	Usability For Handicapped	Usability For APT Resident	Usability For VIPs	Comfort	Minimum Waiting Time	Low Cost	Energy Efficiency
VIP						+				
Call Cancellation					+	-				
Door Time Ext				+						
Reopen by Sensor			+							
Position Sensor	Absolute(XOR)	++							-	
	Relative (XOR)	+							+	
Motor	Hydraulic(XOR)							-		-
	Electric(XOR)							+		+
	Geared(XOR)						-	-	+	-
	Gearless(XOR)						+	+	-	+
Weight Sensor	Load Cell(XOR)						+		-	
	Limit Switch(XOR)								+	
Preemption						+				
Calculation	AbsolutePoistion (OR)	++							-	
	RelativePoistion (OR)	+							+	
Compensation		+							-	
Leveling Profile							+			
	Exponential Profile(XOR)						++			
	Uniform Profile(XOR)						+			
Rope Compensation							+			

In this way, domain knowledge influencing feature selection can be modeled into the three FDs and three mappings between them. The next section describes how the domain knowledge models are used to derive a product feature configuration.

4.4 Derivation of Product Feature Configuration

Feature Configuration starts by selecting usage context features for a particular product. Suppose for example that we select a set of usage context features (*Passenger*, *Office Worker*, *>10*, *Asia*) as a *UC Configuration*. With the *UC Configuration* and the *UC-QA Mapping* in Table 2, we can derive a *QA Configuration* that consists of *Door Safety*, *Usability*, *Minimum Waiting Time*, *Usability for VIPs, Comfort*, and *Position Accuracy*.

The final step is to derive a *PF Configuration* from the *UC Configuration* and the *QA Configuration*, using *QA-PF* and *UC-PF Mappings*. At first, we can eliminate some configuration choices of the *Product FD* by applying the constraints derived from the *UC Configuration* and the *UC-PF Mapping*. In the example, given the *UC Configuration* and the *UC-PF Mapping* in Table 1, we can derive constraints requiring *Electric* and *Gearless* but excluding *60*, *450*, *4500*.

In addition to the constraints, feature selection can be performed based on the assessment results derived from a *QA Configuration* and *QA-PF Mapping*. In this example, given the *QA Configuration* and the *QA-PF Mapping* in Table 3, we can select features (*VIP*, *Reopen by Sensor*, *Absolute*, *Electric*, *Gearless*, *Load Cell*, *Preemption*, *AbsolutePosition*, *Compensation*, *Exponential Profile*, and *Rope Compensation*) whose assessment values are either positive or higher than the other alternatives.

Finally, remaining features, if exist, have to be configured. In case rationales on selecting features have not been captured in the domain knowledge model, we can augment the model with such information.

5 Related Work

This paper is related to researches on modeling the knowledge of an application domain or a product line, and use of the model(s) for product derivation. In software product line engineering, the feature diagram [8] and its extensions [3, 5, 9] have played an important role in documenting not only commonalities and variabilities of a product line but also configuration dependencies as domain knowledge.

In FODA [8] and FORM [9], it is stated that the feature diagram should be augmented with issues and decisions to provide rationales for choosing options and selecting among alternatives. Those issues (e.g., quality attributes) serve as criteria for deciding which features are selected for a product. However, they do not discuss how those issues and rationales are modeled, and how they are related to product features and used in product configuration.

There have been several approaches to modeling quality issues and relating them to features. Yu et al. [16] use a goal model to capture stakeholder goals (e.g., quality attributes) and relate them to features. Thurimella et al. [13] proposed an issue-based variability model that combines the orthogonal variability model (OVM) [11] with a specific rationale-based unified software engineering model (RUSE) [15]. In the model, quality attributes can be modeled as criteria for selecting variant features. Bartholdt et al. [1] and Etxeberria et al. [6] also identified the relevance of quality attributes to the choice of specific features and integrated quality modeling with feature modeling. In contrast to the work by Yu et al. and Thurimella et al., they use a FD to model quality attributes, which is similar to the approach in this paper.

We also model quality attributes with a separate FD (a *Quality Attribute FD*). Correlations between quality attributes and product features are described as the mapping *QA-PF Mapping*. The proposed approach in this paper is mainly different from the work mentioned above in that we made explicit connections to product usage contexts, which, in our experience, mainly decide required quality attributes and drive feature selection.

Recently, several approaches have been proposed to relate product contexts to feature selection. Some of these include the work by Reiser et al. [12], Hartmann et al. [7], and Tun et al. [14].

Reiser et al. use the concept of a *product set* to describe contextual constraints for feature selection. For example, "a certain feature F will be included in all cars for the entire North American market for some reason". This approach is similar to ours in that a group of contextual features (implying multiple products) constrain selection of product features.

Hartmann et al. introduced a context variability model, which is equivalent to the UCVM of this paper, to describe primary drivers for feature selection. A separate FD is used to model contextual variability and linked to a product line FD, which describes the commonality and variability of a product line, through dependency relations such as *requires*, *excludes*, or *restricting cardinality*.

Similarly, Tun et al. describe contextual variability as a separate FD. However, they further separate feature description into three FDs relating to requirements, problem world contexts, and specifications. Two sets of links (between requirement FD and problem world context FD, and between problem world context FD and specification FD) and quantitative constraints are used to describe constraints that affect feature selection. This work is very similar to the proposed approach in this paper, as we also describe domain variability in terms of usage contexts, quality attribute, and product features, which corresponds to problem world contexts, requirements, and specifications, respectively. However, they do not take into account correlations between quality attributes and product features. Moreover, the proposed approach in this paper is mainly different in that we treat product usage contexts as key drivers for deciding product feature configurations.

6 Conclusion

This work was motivated from the observation that products with similar functionalities were often implemented differently. This was mainly because selection of operating environment, domain technology, and implementation features and architectural design decisions (i.e., integration of the selected features) were largely driven by the quality attributes required for a product. In turn, the quality attributes were decided by the context of the product use. This is an important finding in product line engineering. Researches on product configuration has typically focused on functionalities and operating environments, and has not addressed usage context analysis rigorously or as a major analysis subject. However, usage context is the critical driver for product configuration. Understanding their implications on feature selection is considered an essential activity for deriving an optimal feature configuration.

In this paper, we presented a domain knowledge model, which describes not only product line variabilities in terms of product features but also domain knowledge that affects selection of variable product features. The proposed approach separates domain knowledge into three variability models related to usage contexts, quality attributes and product features, and defines mappings between each pair of them.

The domain knowledge model plays a central role in deriving an optimal feature configuration. As described in sections 3 and 4, selection of usage context features in the *Usage Context FD* drives contextual constraints as well as quality attributes. Contextual constraints reduce the configuration space of product features, while quality attributes serve as evaluation criteria for feature selection. If we specify objective functions (e.g., *maximize* or *minimize*) over the quality attributes, we can obtain feature configurations that are optimal with respect to given objective functions. In this paper, correlations between quality attributes and product features are defined as qualitative values, and the work by Bartholdt et al. [1] can complement our work.

The main advantage of our approach is that we can systematically record and relate rationales for feature selection as a coherent model. In case where there are a small number of products with a few contextual variations, constructing this model may not be critical. However, as the scope of a product line is expanded and contextual variations increase, documenting and managing this model become indispensable for deriving valid and optimal feature configurations.

The process of constructing these models is iterative and incremental. For a product feature, we can think about its usage contexts, which may trigger refinement of required quality attributes, which again trigger refinement of the product feature into operating environment and/or implementation features.

As future work we are planning to provide a tool support for constructing the domain knowledge model, deriving feature configurations automatically, or reasoning on model validation. We validated the concept of our approach on a small industrial project in the flash memory domain. To validate scalability, we plan to apply the approach to a larger application domain.

Acknowledgments. This research was supported by the National IT Industry Promotion Agency (NIPA) under the program of Software Engineering Technologies Development.

References

1. Bartholdt, J., Meda, M., Oberhauser, R.: Integrating Quality Modeling with Feature Modeling in Software Product Lines. In: 4th International Conference on Software Engineering Advances, pp. 365–370. IEEE CS, Los Alamitos (2009)
2. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
3. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated Reasoning on Feature Models. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg (2005)
4. Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers, Boston (1999)

5. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration Using Feature Models. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 266–283. Springer, Heidelberg (2004)
6. Etxeberria, L., Sagardui, G.: Variability Driven Quality Evaluation in Software Product Lines. In: 12th International Software Product Line Conference, pp. 243–252. IEEE CS, Washington (2008)
7. Hartmann, H., Trew, T.: Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains. In: 12th International Software Product Line Conference, pp. 12–21. IEEE CS, Washington (2008)
8. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, SEI. Carnegie Mellon University, Pittsburgh, PA (1990)
9. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A Feature-Oriented Reuse Method with Domain Specific Reference Architectures. Ann. Soft. Eng. 5, 143–168 (1998)
10. Mannion, M., Savolainen, J., Asikainen, T.: Viewpoint Oriented Variability Modeling. In: 33rd Annual IEEE International Computer Software and Application Conference, pp. 67–72. IEEE CS, Washington (2009)
11. Pohl, K., Böckle, G., van der Linder, F.: Software Product Line Engineering Foundations, Principles, and Techniques. Springer, Berlin (2005)
12. Reiser, M.O., Weber, M.: Using Product Sets to Define Complex Product Decisions. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 21–22. Springer, Heidelberg (2005)
13. Thurimella, A.K., Bruegge, B., Creighton, O.: Identifying and Exploiting the Similarities between Rationale Management and Variability Management. In: 12th International Software Product Line Conference, pp. 99–108. IEEE CS, Washington (2008)
14. Tun, T.T., Boucher, Q., Classen, A., Hubaux, A., Heymans, P.: Relating Requirements and Features Configurations: A Systematic Approach. In: 13th International Software Product Line Conference, pp. 201–210. Carnegie Mellon University, Pittsburgh (2009)
15. Wolf, T.: Rationale-Based Unified Software Engineering Model. Dissertation Technische Universität München (2007)
16. Yu, Y., Lapouchnian, A.: Configuring Features with Stakeholder Goals. In: The 2008 ACM Symposium on Applied Computing, pp. 645–649. ACM, New York (2008)

A Flexible Approach for Generating Product-Specific Documents in Product Lines

Rick Rabiser[1], Wolfgang Heider[1], Christoph Elsner[2], Martin Lehofer[3], Paul Grünbacher[1], and Christa Schwanninger[2]

[1] Christian Doppler Laboratory for Automated Software Engineering, Johannes Kepler University, Linz, Austria
rabiser@ase.jku.at
[2] Siemens Corporate Research & Technologies, Erlangen, Germany
christa.schwanninger@siemens.com
[3] Siemens VAI Metals Technologies, Linz, Austria
martin.lehofer@siemens.com

Abstract. In software product line engineering various stakeholders like sales and marketing people, product managers, and technical writers are involved in creating and adapting documents such as offers, contracts, commercial conditions, technical documents, or user manuals. In practice stakeholders often need to adapt these documents manually during product derivation. This adaptation is, however, tedious and error-prone and can easily lead to inconsistencies. Despite some automation there is usually a lack of general concepts and there are "islands of automation" that are hardly integrated. Also, research on product lines has so far often neglected the handling of documents. To address these issues, we developed a flexible approach for automatically generating product-specific documents based on variability models. We applied the approach to two industrial product lines of different maturity using the decision-oriented product line engineering tool suite DOPLER.

Keywords: document generation, model-based approach, product lines.

1 Introduction and Motivation

Software product line engineering (SPLE) [1, 2] traditionally has a strong focus on technical software assets such as architecture or code. Researchers and practitioners are typically adopting feature models [3], decision models [4], UML-based techniques [5], or orthogonal approaches [1] to define the reusable assets' variability. Often these approaches are used to expedite and automate product derivation, e.g., by generating configurations. However, in practice documents play an equally crucial role. Customers, sales people, marketing staff, or product managers frequently work with documents such as offers, user manuals, commercial conditions, or contracts. Also, developers or testers need to provide documents as part of their daily work, for example technical documentation or test plans.

In our collaboration with industry partners in different domains, we have learned that while the derivation and configuration of products is often at least partly automated,

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 47–61, 2010.

documents are typically still adapted manually. For example, sales people customize offers and user manuals to be delivered to customers. Engineers adapt technical documentation for customer-specific development and maintenance. This manual adaptation is often tedious and error-prone and the created documents can easily become inconsistent with the derived software system.

Researchers have presented approaches for extracting reusable assets and variability information from legacy documentation to support defining a product line [2, 6]. There also exists work on developing documentation for software product lines and more generally on reusing such documentation [7, 8]. However, regarding variability modeling and product derivation most research in SPLE focuses on technical software assets and treats documents rather as a side issue. Basic support for modeling document variability and adapting documents is for instance available as part of a commercial SPLE tool [9]. Technically, adapting documents seems straightforward. Concepts from model-driven SPLE [10, 11] can for example be applied to structured documents. However, a generic and flexible approach is still missing that provides concepts for automating document generation independently of the concrete type of documents and regardless of the maturity of the product line, i.e., the degree to which variability modeling and generative techniques are already applied.

Based on an existing model-driven SPLE approach [12, 13] we developed an approach that supports the generation of both deliverable documents and software systems from the same variability model. The approach is based on the observation that many decisions made during product derivation are relevant to both types of assets. For example, the decision to include a certain feature will also require the inclusion of the user documentation of this feature. Decisions made in product derivation typically also have an effect on sales documents such as offers or contracts. For example, descriptions of selected features must also be part of such documents.

Using the tool suite DOPLER [12, 13] we successfully applied our approach in the domain of industrial automation to two product lines of different maturity:

The first application example is a mature software product line for process automation of continuous casting machines in steel plants from Siemens VAI. The architecture of this software is modularized and built as Java Beans using the Spring component framework [14]. In this case the variability has already been modeled for the technical software assets [12, 15] and configuration files for concrete solutions can be generated using DOPLER. Examples for variability in a continuous caster delivered by Siemens VAI are the number of strands, the choice of whether to include a cooling system, and the type of cooling. Such variability not only affects technical software assets but also directly shapes the user documentation of the software. Interviews with domain experts revealed that the manual adaptation of documentation for each customer (300+ pages) can be tedious and automation would be highly beneficial.

The second industrial application example deals with automating the generation of sales documents like customer-specific offers, product descriptions, and commercial condition documents at Siemens AG for a family of electrode control systems for electric arc furnaces (EAF) in steel production. The corresponding sales department has to provide many specific documents to prospective customers. Creating these documents requires the sales people to “parse” and “process” the documents manually each time a quote has to be submitted. Erroneous offers may result in actual losses or legal issues; therefore the manually created documents are thoroughly reviewed and

checked for quality. This altogether can extend the duration for creating offers to several weeks. Currently the offer process takes two weeks in average and 120 offers are written a year. The time between the initial customer contact and the delivery of the offer matters a lot to customers. The goal is thus to reduce the time needed for an offer to at most one week to improve customer satisfaction. Fortunately, in this industrial case the variability in the documents has already been known to a large extent. Many of the documents already contained explicit variability information about product features, target environment, and commercial and legal conditions.

Our experience shows that in the first case of a mature software product line it is possible to largely reuse the existing variability models for generating documents. In our second application example automated product derivation techniques have not yet been applied. In this case, however, document generation provides a convincing showcase for the benefits of variability analysis, modeling, and automation. In both projects decision models were considered very helpful for describing variability, especially by non-technical stakeholders.

The remainder of this paper is structured as follows: We first give an overview of our flexible document generation approach. We then present the concrete technical realization based on DOPLER. We illustrate the feasibility of the approach by providing details of applying it in the two industrial application examples. We discuss related work and conclude the paper with an outlook on future work.

2 Approach

Our approach for document generation in product derivation is independent of the maturity of the product line, the concrete variability modeling technique, and the type of documents to be generated. Existing model-driven SPLE techniques [10, 11, 13] can for example be used for implementation if they support working with documents. The approach comprises four steps which are usually conducted iteratively:

(1) *Elicit and analyze variability in documents.* An industrial product line rarely starts from scratch, in particular when considering automation. Examining existing documents such as user manuals or contracts helps to reveal variability. Therefore a product line expert familiar with variability modeling needs to analyze such existing assets. Additionally, it is advisable to conduct workshops with domain experts from product management, sales, and development. Such experts have frequently been dealing with variability in documents in the past and they know which manual adaptations have been most relevant [6]. The document analysis shows what can vary (variation points) and how it can vary (variants). The key challenge is to find the right level of detail and granularity. It does not make sense to elicit and analyze all possible variability. Instead, domain experts should focus on the most relevant variability that can potentially attain the highest cost savings.

(2) *Create or adapt variability models.* The product line expert uses the collected information to either adapt existing models or to create new ones. If existing models already guide and automate product derivation, a considerable part of the models might also be used for automating document generation. For instance, the selection of a particular feature in product derivation might not only require the inclusion of a software component, but also of related documents. There are no restrictions regarding the technique

used to model document variability. It should however be flexible and should allow advanced automation during product derivation.

(3) *Choose or develop a variability mechanism and a corresponding generator for domain-specific document formats*. A generator is required that automates the creation of product-specific documents according to variant selections. Such a generator typically relies on explicitly defined variation points in the documents. The mechanisms used to define variation points depend on the documents' formats. Some unstructured formats (e.g., txt, rtf) are not well suited for that purpose. Documents thus need to be converted to formats for which document processors either exist [16] or can be developed with reasonable effort. For example, document formats of popular office suites like Microsoft Word can be extended using markups [9]. Beyond a variability markup mechanism it is useful to partition large documents and to add meta-information for coping with coarse-grained variability. Many existing document assembly tools use XML-based formats for that purpose.

(4) *Augment the documents with variability information.* Product line experts formally specify the variation points and variants from step (1) in the documents and relate them to the variability models from step (2) using the mechanism chosen in step (3). During product derivation the generator then resolves variation points according to a particular input configuration and produces product-specific documents.

3 Tool Architecture and Realization

We implemented the described approach using existing technologies and tool suites. For variability modeling we use the decision-oriented DOPLER approach [12, 15]. For defining variability in documents we adopted and extended the DocBook system [16] and developed a generator extension for DOPLER [13].

3.1 The DOPLER Approach and Tooling

DOPLER is a decision-oriented SPLE approach [12, 15] comprising a variability modeling tool and a configuration wizard to present variability to users in product derivation. DOPLER variability models contain Assets and Decisions (cf. upper part of Fig. 1). *Assets* represent the core product line artifacts (e.g., technical components or documents). Assets can depend on each other functionally (e.g., one component requires another component) or structurally (e.g., a paragraph is part of a chapter). DOPLER allows modeling assets at arbitrary granularity and with domain-specific attributes and dependencies, based on a given set of basic types. Users can create domain-specific meta-models to define their asset types, attributes, and dependencies.

In DOPLER variation points are defined and presented as *decisions*. Important attributes of decisions are a unique id, a question that is asked to a user during product derivation, and a decision type (Boolean, enumeration, string, or number). A decisions can depend on other decisions hierarchically (if it needs to be made before other decisions) or logically (if making the decision changes the answers of other decisions). The decision type describes the range of possible answers and can be further constrained with validity conditions. Decision models have proven useful in application engineering as they allow describing variability at a higher level of abstraction

matching the problem space terminology of domain experts [17]. Decision models in conjunction with asset models also increase flexibility, as a single decision model can be used with several asset models defining different types of assets such as components or documents. In DOPLER, assets are linked to decisions via inclusion conditions defining when a particular asset is included in a derived product. Asset attributes can also depend on answers to decisions to enable the customization of assets.

3.2 Modeling Documents and Their Variability with DOPLER

DOPLER allows defining domain-specific asset types by creating a meta-model for a particular organization or context. We can thus include documents or parts of documents as dedicated assets in our models. We have defined the generic asset type `document fragment` representing arbitrary parts of documents, such as chapters or sections. Each `document fragment` has an attribute `location` (of type URL) in addition to the default model attributes `id` and `description`. We also defined that `document fragments` can be part of other fragments, e.g., a section might be part of a chapter, and that they can contribute to other assets, e.g., if a fragment describes a particular component. The generic asset type `document fragment` can be further refined with additional attributes and relations for domain-specific purposes. Fig. 1 shows the DOPLER meta-model extended with `document fragments`.

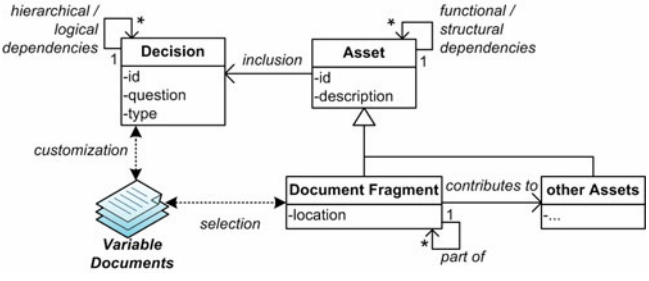


Fig. 1. Extended DOPLER meta-model with documents

Decisions represent document variability as questions that a user is expected to answer during product derivation. `Document fragment` assets represent arbitrary parts of documents and are used to model coarse-grained variability. Explicitly representing parts of documents as assets in a model provides an additional level of abstraction for defining document variability and for selecting a particular chapter or section for inclusion in derived documents. It would also be possible to describe document variability only at the level of decisions (and use the answers to decisions to customize documents *and* select parts of documents). However, we have learned that the additional level of abstraction helps domain experts in understanding coarse-grained document variability. For example, the `document fragment` asset *coolingchapter* represents the documentation of the cooling system and depends on the answer to decision "*Shall the cooling system be delivered?*". As soon as a user answers the question with *yes*, the asset *coolingchapter* is marked to be included in the derived product to resolve this coarse-grained variability. The same decision often can be used for

several *document fragments* or for including both documents and components thus making the approach more flexible. To support fine-grained customizations, we developed a variability mechanism based on DocBook.

3.3 A Variability Extension for DocBook

DocBook [16] is a collection of standards and tools for technical publishing proposed by a consortium of software companies as a computer documentation standard. The core of DocBook is a Document Type Definition (DTD) defining content elements and their possible relations that authors can use for creating documents. For example, the *book* element can contain a *title* element which can contain *para* and *chapter* elements. Using the DTD and XML syntax, authors can write text content with markup using arbitrary XML tools. Components for editing, typesetting, and version control can be combined as needed and a set of XSL style sheets is available for converting content to arbitrary target formats such as HTML or PDF. DocBook is well-suited for automatically processing documents (see e.g., [16, 18]). It can be compared with LaTeX but uses XML markup.

DocBook provides a profiling mechanism for extending the DTD by defining new elements and attributes. We use this mechanism to define elements and attributes for implementing variation points in documents. For instance, we defined an attribute *doplerdoc* that can be used with all elements in DocBook files. The value of the *doplerdoc* attribute can refer to the unique id of a decision or an asset (i.e., *document fragment* asset). Adding the *doplerdoc* attribute, e.g., to a *chapter* or *para* element, allows tagging it as optional/alternative depending on whether a particular asset is included in a derived product. The element *doplerdocplaceholder* can be used in combination with the *doplerdoc* attribute to define a placeholder in the text that will be filled with the answer to a particular decision (cf. Listing 1).

Listing 1. An example showing how DocBook files can be parameterized to make them variable and relate them to a DOPLER variability model

```
<!--chapter "cooling" is included in the documentation of the product if document fragment
asset "coolingchapter" is included due to some decision made in product derivation.-->
<chapter id="cooling" doplerdoc="coolingchapter">
      <!-- paragraph "cooling_mechanism" includes text with a placeholder. This placeholder
      is replaced with the answer set to decision "cooling_mech" in product derivation. -->
      <para id="cooling_mechanism">
      For secondary cooling, the caster supports the
      <doplerdocplaceholder doplerdoc="cooling_mech"/> mechanism. ...
      </para>
      <para id="...">...</para>
</chapter>
```

We analyzed the technical user documentation as well as sales documents (i.e., offers and commercial conditions) in the domain of industrial automation systems (cf. Section 4) to identify common patterns of variability in documents. Table 1 presents the most common and relevant types we found, provides examples, and demonstrates how the different types of variability can be implemented.

Table 1. Implementing document variability using DocBook and DOPLER

Document Variability	**Description and Example**	**Implementation in DocBook**

Placeholders	Items that are replaced with text or values that are entered and calculated during product derivation, e.g.,
	- customer details (name, company, address, etc.),
	- total price,
	- return on investment values.
		Use designated *doplerdocplaceholder* element and use a *doplerdoc* attribute to relate the placeholder with a decision, e.g.,
		`<section id="caster">`
		`The caster has`
		`<doplerdocplaceholder`
		`doplerdoc="numStrands"/>`
		`strands. </section>`

Optional text	Chapters or sections, paragraphs, sentences, or even words or letters are added or removed depending on the features to be delivered, e.g., chapters in a user manual or in a bidding document.	Mark DocBook element as optional using the *doplerdoc* attribute to relate it with a *document fragment* asset, e.g.,
		`<chapter id="hmi"`
		`doplerdoc="hmichapt">`
		`... </chapter>`

Alternative text	Chapters or sections, paragraphs, sentences, or even words or letters that alternate depending on the target market, customers and system environment, e.g.,
	- country-specific commercial conditions and policies,
	- different operating systems,
	- units (e.g., metric vs. imperial system).
		Enclose alternative parts with DocBook elements and add *doplerdoc* attributes related with *document fragments*, e.g.,
		`<section id="dex"`
		`doplerdoc="dexchapter">`
		`Data Exchange is supported via`
		`<phrase doplerdoc="asciiphrase">`
		`ASCII</phrase>`
		`<phrase doplerdoc="dbphrase">`
		`DataBase</phrase>`
		`<phrase doplerdoc="tcpipphrase">`
		`TCP/IP</phrase>`
		`in your system. ...</section>`

Cross references	References to document internals and links to external documents not included in the generated document must be found and replaced to avoid tangling references, e.g.,
	- the main index,
	- figure and table indices,
	- references to docs like country-specific legal documents.
		References and links in DocBook must be related to the same *document fragment* assets as the parts they reference, e.g.,
		`<xref linkend="hmi"`
		`doplerdoc="hmichapt"/>.`
		
		If additional text describes the link (e.g., "...refer to chapter..."), this text must also be enclosed with an XML element dependent on the *document fragment* asset.

Simple grammatical variability	Mainly regards singular vs. plural but also gender, e.g.,
	- 1 strand vs. 2 or more strands,
	- multiple drives vs. the drive.
		Respective text has to be enclosed with elements and marked with the *doplerdoc* attribute. An additional dependency to a numerical decision allows defining whether to use singular or plural, e.g.,
		`The caster has`
		`<doplerdocplaceholder`
		`doplerdoc="numStrands"/>`
		`strand<phrase`
		`doplerdoc="numStrands#2+">`
		`s</phrase>.`

Table 1. (*Continued*)

Media objects	Media objects (e.g., images) might have to be replaced depending on sales aspects and the delivered system, e.g.,
	- customer's logo,
	- user interface.
		Mark DocBook element as optional using designated `doplerdoc` attribute, e.g.,
		`<mediaobject doplerdoc="hmichapt">`
		`...`
		`</mediaobject>`

Formatting/ Layout	Different styles might be required/desired for different users, e.g., - A4 vs. letter size, - different color schema.	Formatting/Layout is achieved using XSL transformation and CSS style sheets. The selection which XSL file and/or CSS style sheet is used can be related to a property file which can be generated based on decisions.

3.4 Generating Documents Using DocBook

Fig. 2 depicts an overview of our tool architecture. The DOPLER tool suite supports defining document variability in models, augmenting documents with variability information, as well as communicating document variability to end-users using the configuration wizard (cf. Fig. 3 in Section 4.2). Traceability from the model's assets and decisions to DocBook is achieved via the dedicated XML attribute `doplerdoc` and XML element `doplerdocplaceholder`.

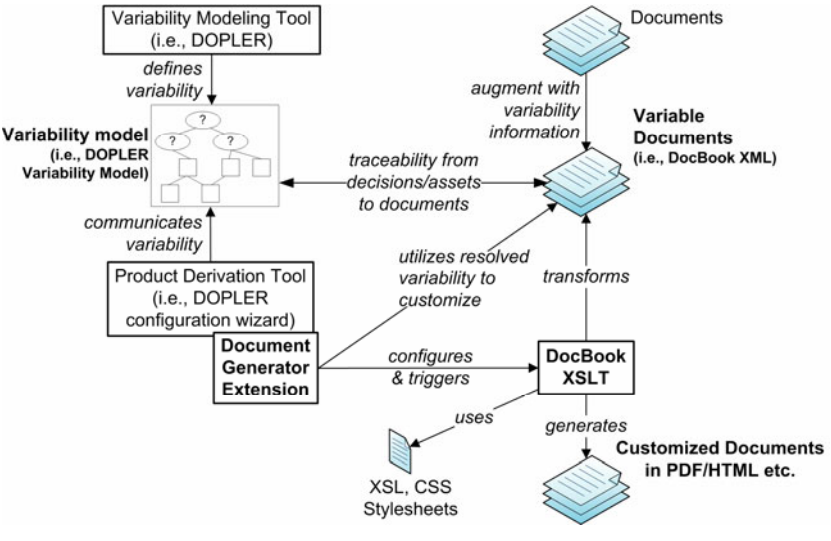


Fig. 2. Architecture for managing and customizing documents in a product line

The configuration wizard can be extended with domain-specific generators. We developed a generator that uses the answers to decisions and the selected `document fragment` assets to compose documents. The generator customizes the documents using answers to decisions and post-processes the documents using DocBook's XSLT

engine and style sheets (which themselves can depend on decisions). Different output formats can be generated (e.g., PDF or HTML).

4 Industrial Application Examples

We applied the described approach in two industrial product lines with different maturity. In a project with Siemens VAI variability models describing a continuous casting automation software product line were already available and generating product configurations was already supported. The goal in this project was to also support generating technical user documentation. In a second project with Siemens AG, we applied the document generation approach to the EAF product line where no variability modeling has been used before. The goal of this second project was to support the automated generation of customer-specific offers, product descriptions, and commercial condition documents.

4.1 Industrial Application Example I: Generating Technical User Documentation for a Continuous Casting Automation Software Product Line

In the project with Siemens VAI we applied our approach with the goal of automating the generation of customer-specific technical user documentation by reusing the decisions modeled for software configuration. Parts of the technical user documentation were already available as DocBook sources which made adding variability meta-information pretty straightforward.

(1) *Elicit and analyze variability in documents*. Based on interviews with domain experts, we identified relevant variability in the documentation (cf. Table 1). According to the domain experts, the following variation points occur frequently when adapting the documentation for a particular customer: (i) parts of the documentation are optional depending on the parts of the system to be delivered, (ii) cross references have to be adapted to avoid tangling references when particular sections are not shipped (the same applies to indices), (iii) numerous placeholders in the text need to be replaced (e.g., customer name), (iv) grammatical changes have to be performed (e.g., strand vs. strands), and (v) specific documents need to be deployed if customers intend to develop extensions to the system instead of using it "out of the box".

(2) *Create or adapt variability models*. We extended the generic DOPLER metamodel for modeling `document fragment` assets as discussed in Section 3.2. We captured the diverse parts of the documentation as `document fragment` assets and related these assets to the decisions. In about 70% of the cases we just used software configuration decisions already defined in the existing variability models. In the remaining cases we added new decisions specifically for document generation.

(3) *Choose or develop a variability mechanism and a corresponding generator for the domain-specific document format*. We developed a generator extension for the DOPLER configuration wizard that reassembles the technical user documentation according to the decisions for the derived product (cf. Section 3.4). This generator resolves the variability and uses libraries for performing the proper transformations from the DocBook sources into the selected target format (e.g., HTML or PDF).

(4) *Augment the documents with variability information.* We augmented the existing DocBook sources with the elicited variability, i.e., for every kind of variability mentioned by the domain experts, we implemented at least one example to demonstrate the feasibility of our approach (cf. Table 1). These examples were sufficient to enable the domain experts to annotate the rest of the documents themselves.

The resulting tool chain supports modeling the variability of the software and the documents in an integrated manner and automates the generation of both product configurations and technical user documentation using a single decision model.

4.2 Industrial Application Example II: Generating Sales Documents for an Electrode Control System for Electric Arc Furnaces (EAF)

In the EAF project we applied our approach to support the automated generation of customer-specific sales documents. When using the DOPLER configuration wizard variability can be resolved in interactive interviews. Prospective customers and sales people jointly answer the questions defined in the decision model (cf. Fig. 3). In case the customer can provide all the data required, several documents can be generated immediately, e.g., offers, product descriptions, return-on-invest (ROI) estimations, and commercial conditions. Special customer requests need to be treated manually but can be captured using the configuration wizard tool [15].

(1) *Elicit and analyze variability in documents*. Together with domain experts, we analyzed existing Microsoft Word documents for offers, price lists, product descriptions, and commercial conditions to identify commonalities and variability. According to the domain experts, the following variability occurs frequently when adapting the documents for a particular customer: (i) *in offers* the customer details, price lists, total prices, and ROI values differ depending on the customer and selected features (i.e., "sellable units"); (ii) *in product descriptions* the component description chapters differ depending on the selected features of the product; (iii) *in commercial conditions* the values and text with regard to the payment method, paying installments, delivery details, warranty conditions, currency, and validity of the offer can differ. Many variation points were obvious (e.g., price list items) and some variability was already tagged with comments within the Word documents. Some variation points span more than just one document. For example, at customization time, the offer is extended with the offered items of the price list and values for the total price as well as the ROI are calculated. The product description document is extended with additional chapters for every additional system component.

(2) *Create or adapt variability models.* We slightly adapted the DOPLER metamodel (cf. Fig. 1 in Section 3.2) for dealing with sales documents. The parts of documents defined in the model represent sellable units. Thus, we called the asset type representing document fragments `feature` in this case (not to be confused with the notion of a feature in feature modeling!). `Feature` assets have the attribute `price`. This is required for the generator which in this case calculates values like the total price and the ROI besides generating documents. We modeled decisions for capturing customer details (e.g., name and address), the system environment (e.g., voltage of available power supply), and values needed for ROI calculations (e.g., price for electricity). In total we extracted 101 decisions for deriving the complete set of documents

with all variants for prospective customers. These decisions are hierarchically organized and numerous interdependencies are defined.

(3) *Choose or develop a variability mechanism and a corresponding generator for the domain-specific document format*. We reused the generator extension developed for Siemens VAI and extended it for generating offers. For business values like the total price and ROI the generator performs the necessary calculations and saves the results to text files. The content of these files is referenced in the DocBook sources to include these values in the generated documents.

(4) *Augment the documents with variability information*. Like for Siemens VAI we implemented the variation points in DocBook sources we created manually from the existing Word documents. While we could also have augmented the Word documents directly (e.g., using markups [9]), our industry partner decided to use DocBook. Within the DocBook source files we applied the described document variability concepts (cf. Table 1), such as placeholders, optional parts, alternatives, and cross references to enable the automated generation of sales documents.

Fig. 3 shows the configuration wizard generating offers for EAF.

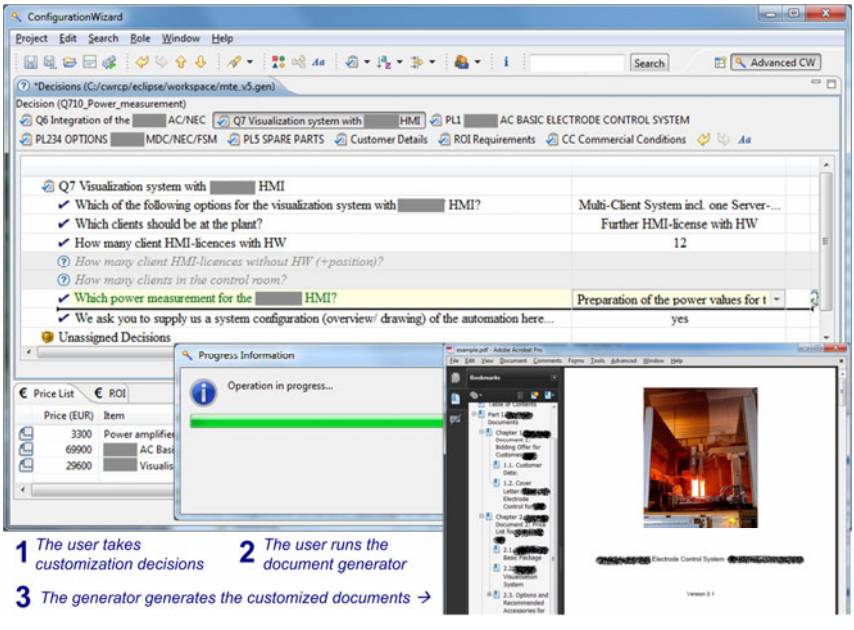


Fig. 3. Making decisions in the DOPLER derivation tool configuration wizard and generating sales documents in the EAF example (partly blurred due to non-disclosure agreements)

The user can make customization decisions by answering questions. The document generator can be started at any time. Based on the user's answers to questions the tool computes the required list of document assets. The generator then creates customized documents in PDF format based on this list.

5 Related Work

We present related work grouped in two areas of research, i.e., research on model-driven product derivation and research on modeling and generating documents in a product line context.

Model-driven product derivation. Several approaches to model-driven product derivation in SPLE have been proposed using a variety of concepts and technologies to support model transformations and product generation. For example, Groher and Völter [10] integrate concepts from model-driven and aspect-oriented software development to support model composition and transformation in an SPLE context. Another example is the work by Ziadi and Jézéquel [11], who present an approach using UML model-transformation to support product derivation in the context of SPLE with UML. Sánchez *et al.* [19] propose VML4*, a tool-supported process for specifying product derivation processes in SPLE that automates product derivation with model transformations. Similar tools have been proposed, e.g., FeatureMapper [20] or CVL [21]. These and other existing model-driven product derivation approaches and tools however have so far focused on technical software assets (mainly software architecture) and treat documents as a side issue or not at all. We believe that many of the existing model-driven approaches and tools could also be extended to support modeling document variability and document generation, similar to what we did in this paper with DOPLER. However, the approaches are not well suited for sales experts and other non-technical stakeholders.

Modeling and generating documents in a product line context. Nicolás and Toval [22] present a systematic literature review on the generation of textual requirements specifications from models in software engineering. The review shows that a lot of work exists on generating requirements specifications from models of different kinds. However, there is a lack of support for modeling and generating documents of different types in SPLE. Dordowsky and Hipp [8] report on the introduction of SPLE for an avionics system where they also had to consider documents (i.e., software design and interface design descriptions) and their generation. Their domain-specific solution however does not seem to be generally applicable for modeling and generating documents in SPLE. Koznov and Romanovsky [7] propose DocLine, a method for developing documentation for software product lines that allows reusing document fragments. They also propose a "documentation reuse language" for defining documentation. They focus on documentation for the product line and derived products but do not specifically address generating arbitrary other kinds of documents like offers. Gears [23] includes document adaptation in an approach manipulating all kinds of product line artifacts. The company pure-systems describes how to use its commercial variability management tool pure::variants to customize Microsoft Word documents depending on features selected in a feature model [9]. With our approach, we do not focus on one specific format of documents (DocBook could be replaced quite easily by developing a new generator) and suggest to explicitly define coarse-grained document variability in models (*document fragment* assets). Most importantly, we use end-user customization tools based on decision models to resolve variability.

6 Conclusions and Future Work

The main contributions of this paper are: (1) a flexible, tool-supported approach to model the variability of documents in SPLE and end-user support for generating product-specific documents in product derivation and (2) an initial evaluation based on two industrial examples where both technical as well as business documents are important. We implemented the approach using the DOPLER and DocBook tool suites. However, a different variability modeling approach and document format might be used for the same purpose. Especially if converting the source documents to DocBook is infeasible in an organization – due to internal regulations – other formats have to be extended to express variability like in [9]. The focus of our approach is on flexibility with regard to possible document types and different types of variability in documents. Our extensible tool support allows development and integration of arbitrary generators for generating documents based on a variability model. Moreover, the questionnaire-like structure of DOPLER decision models supports stakeholders without technical background knowledge.

Both industrial partners plan to integrate our approach in their process and tool landscape. We will support pilot projects to further improve our approach and tools. The extensibility of our tools will enable developing extensions for the integration of our tools in the organizations' tool environment. For instance, one of our partners uses a proprietary tool for generating technical specifications that could substitute DocBook. Furthermore existing document management systems and collaboration infrastructure will be used for managing standard text that is subject to changes regularly, such as text for legal conditions. This will prevent errors in the offer process and helps keeping review cycles short. In the EAF example the results will be used as a first step to further automate product derivation. For example, the layout of the Electrode Control System and the software configuration might also be generated from the same decision model.

We plan to perform further case studies in other domains to validate our approach and identify useful process automations, e.g., the automatic conversion of Microsoft Word documents to DocBook. We refer to unique id's of model elements in documents which can complicate evolution of models and documents. We have been developing incremental consistency checking support for code assets [24] and plan to adapt it for documents in our future work. Furthermore, we want to study more complex settings like multi-stage configuration. Sales people, technicians, and lawyers have to work together to get all the decisions right [25]. Similarly, the collaboration of a headquarter with regional organizations complicates matters. Some decisions might depend on the region, while others are to be decided by the headquarter.

Acknowledgments. This work has been supported by the Christian Doppler Forschungsgesellschaft, Austria and Siemens VAI Metals Technologies as well as Siemens Corporate Research & Technologies. We thank Manuel Wallnöfer who contributed to the development of a document generator for one industrial application.

References

1. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
2. van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action -The Best Industrial Practice in Product Line Engineering. Springer, Heidelberg (2007)
3. Kang, K.C., Lee, J., Donohoe, P.: Feature-Oriented Product Line Engineering. IEEE Software 19(4), 58–65 (2002)
4. Schmid, K., John, I.: 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)
5. Gomaa, H.: Designing Software Product Lines with UML. Addison-Wesley, Reading (2005)
6. Rabiser, R., Dhungana, D., Grünbacher, P., Burgstaller, B.: Value-Based Elicitation of Product Line Variability: An Experience Report. In: Heymans, P., Kang, K.C., Metzger, A., Pohl, K. (eds.) Proceedings of the Second International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2008), Essen, Germany, pp. 73–79. University of Duisburg Essen (2008)
7. Koznov, D.V., Romanovsky, K.Y.: DocLine: A method for software product lines documentation development. Programming and Computer Software 34(4), 216–224 (2008)
8. Dordowsky, F., Hipp, W.: Adopting Software Product Line Principles to Manage Software Variants in a Complex Avionics System. In: McGregor, J.D. (ed.) Proceedings of the 13th International Software Product Line Conference (SPLC 2009), San Francisco, CA, USA. ACM International Conference Proceeding Series, vol. 446, pp. 265–274. Carnegie Mellon University, Pittsburgh (2009)
9. Pure:systems GmbH: Automatic Generation of Word Document Variants (2010), `http://www.pure-systems.com/flash/pv-word-integration/flash.html`
10. Voelter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: Proceedings of the 11th International Software Product Line Conference (SPLC 2007), Kyoto, Japan, pp. 233–242. IEEE Computer Society, Los Alamitos (2007)
11. Ziadi, T., Jézéquel, J.M.: Software Product Line Engineering with the UML: Deriving Products. In: Käkölä, T., Duenas, J. (eds.) Software Product Lines, pp. 557–588. Springer, Heidelberg (2006)
12. Dhungana, D., Grünbacher, P., Rabiser, R., Neumayer, T.: Structuring the Modeling Space and Supporting Evolution in Software Product Line Engineering. Journal of Systems and Software 83(7), 1108–1122 (2010)
13. Dhungana, D., Rabiser, R., Grünbacher, P., Lehner, K., Federspiel, C.: DOPLER: An Adaptable Tool Suite for Product Line Engineering. In: Proceedings of the 11th International Software Product Line Conference (SPLC 2007), Kyoto, Japan, vol. 2, pp. 151–152. Kindai Kagaku Sha Co. Ltd. (2007)
14. Johnson, R., Höller, J., Arendsen, A.: Professional Java Development with the Spring Framework. Wiley Publishing, Chichester (2005)
15. Rabiser, R., Grünbacher, P., Dhungana, D.: Supporting Product Derivation by Adapting and Augmenting Variability Models. In: Proceedings of the 11th International Software Product Line Conference (SPLC 2007), Kyoto, Japan, pp. 141–150. IEEE Computer Society, Los Alamitos (2007)
16. Stayton, B.: DocBook XSL. Sagehill Enterprises (2005)

17. Grünbacher, P., Rabiser, R., Dhungana, D., Lehofer, M.: Model-based Customization and Deployment of Eclipse-Based Tools: Industrial Experiences. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), Auckland, New Zealand, pp. 247–256. IEEE/ACM (2009)
18. Walsh, N., Muellner, L.: DocBook: The Definitive Guide. O'Reilly, Sebastopol (1999)
19. Sánchez, P., Loughran, N., Fuentes, L., Garcia, A.: Engineering Languages for Specifying Product-Derivation Processes in Software Product Lines. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 188–207. Springer, Heidelberg (2009)
20. Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: mapping features to models. In: Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, pp. 943–944. ACM, New York (2008)
21. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G., Svendsen, A.: Adding Standardized Variability to Domain Specific Languages. In: Geppert, B., Pohl, K. (eds.) Proceedings of the 12th International Software Product Line Conference (SPLC 2008), Limerick, Ireland, pp. 139–148. IEEE Computer Society, Los Alamitos (2008)
22. Nicolás, J., Toval, A.: On the generation of requirements specifications from software engineering models: A systematic literature review. Information and Software Technology 51(9), 1291–1307 (2009)
23. Krueger, C.: The BigLever Software Gears Unified Software Product Line Engineering Framework. In: Geppert, B., Pohl, K. (eds.) Proceedings of the 12th International Software Product Line Conference (SPLC 2008), Limerick, Ireland, Lero, vol. 2, p. 353 (2008)
24. Vierhauser, M., Dhungana, D., Heider, W., Rabiser, R., Egyed, A.: Tool Support for Incremental Consistency Checking on Variability Models. In: Benavides, D., Batory, D., Grünbacher, P. (eds.) Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010), Linz, Austria, ICB-Research Report No. 37, pp. 171–174. University of Duisburg Essen (2010)
25. O'Leary, P., Rabiser, R., Richardson, I., Thiel, S.: Important Issues and Key Activities in Product Derivation: Experiences from Two Independent Research Projects. In: McGregor, J.D. (ed.) Proceedings of the 13th International Software Product Line Conference (SPLC 2009), San Francisco, CA, USA. ACM International Conference Proceeding Series, vol. 446, pp. 121–130. Carnegie Mellon University (2009)

Formal Definition of Syntax and Semantics for Documenting Variability in Activity Diagrams*

André Heuer[1], Christof J. Budnik[2], Sascha Konrad, Kim Lauenroth[1], and Klaus Pohl[1]

[1] Paluno – The Ruhr Institute for Software Technology, University of Duisburg-Essen, Gerlingstraße 16, 45127 Essen, Germany
{andre.heuer,kim.lauenroth,klaus.pohl}@paluno.uni-due.de
[2] Siemens Corporate Research
755 College Road East, Princeton, NJ 08540, USA
christof.budnik@siemens.com
konradsa@gmail.com

Abstract. Quality assurance is an important issue in product line engineering. It is commonly agreed that quality assurance in domain engineering requires special attention, since a defect in a domain artifact can affect several products of a product line and can lead to high costs for defect correction. However, the variability in domain artifacts is a special challenge for quality assurance, since quality assurance approaches from single system engineering cannot handle the variability in domain artifacts. Therefore, the adaptation of existing approaches or the development of new approaches is necessary to support quality assurance in domain engineering.

Activity diagrams are a widely accepted modeling language used to support quality assurance activities in single system engineering. However, current quality assurance approaches adapted for product line engineering using activity diagrams are not based on a formal syntax and semantics and therefore techniques based on these approaches are only automatable to a limited extent.
In this paper, we propose a formal syntax and semantics for documenting variability in activity diagrams based on Petri-nets which provide the foundation for an automated support of quality assurance in domain engineering.

Keywords: Variability, variability in activity diagrams, documenting variability, quality assurance for product line engineering.

1 Introduction

Early quality assurance is an important issue in every development project [2] and especially in product line engineering [17]. The development process for product lines is separated into domain engineering and application engineering [22]. Therefore, quality assurance is conducted twice. In domain engineering, the quality of domain (or reusable) artifacts (e.g., requirements, design, or implementation artifacts) has to

* This paper was partially funded by the DFG, grant PO 607/2-1 IST-SPL.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 62–76, 2010.

be ensured. In application engineering, the quality of each derived product of the product line has to be assured. It is commonly agreed that quality assurance in domain engineering requires special attention, since a defect in a domain artifact can affect several products of a product line and can lead to high costs for defect correction [17, 22].

During domain engineering, the domain artifacts of the product line are developed. These domain artifacts contain variability, i.e., they contain modification possibilities to address different customer needs [22]. Variability in domain artifacts increases complexity, since a variable domain artifact represents not only a single artifact, but a set of artifacts (e.g., a variable requirement represents a set of requirements) [12]. Due to this complexity, quality assurance for product line engineering is a challenging task, since approaches for quality assurance from single system engineering cannot simply be applied to domain artifacts without adaptation [14].

In this paper, we focus on UML activity diagrams [19], a modeling language that is used in several control flow-based quality assurance approaches for single system engineering (cf. e.g., [3]) and for product line engineering (cf. e.g., [24, 25]). For quality assurance approaches, it is important to represent the control flow of the system under test in a model, since test strategies and the derivation of test cases in single system engineering are based on control flow models [2].

In the following, we illustrate why test approaches from single system engineering cannot be applied to activity diagrams that include variability with a simplified example. Fig. 1 shows an activity diagram that contains variability and a variability model based on Boolean expressions[2]. The edges of the activity diagram are related to variants of the variability model to document that an edge (and related activities) is variable and thus selectable for a derived product. For example, if variant v_1 is selected, only the related edges become part of an application activity diagram and the edges related to variant v_2 would be deleted. Assume, for example, an automated approach for deriving test cases based on a statement coverage criterion [3]. This criterion is based on the coverage of all statements, i.e. every statement has to be processed at least once. If such an algorithm is executed on the domain model shown in Fig. 1, a resulting test case scenario would be the flow 'A2 → A1 → A3'. Taking only the activity diagram into account, this test case scenario is acceptable, since the activity diagram allows this scenario. However, if the variability model is taken into account, the proposed path is not possible, since the selection of edges contradicts the variability model. It requires selecting the variants v_1 and v_2 at the same time, which is not allowed due to the variability model (v_1 XOR v_2).

The only valid approach for applying quality assurance techniques from single system engineering is the derivation of sample products [17]. A derived sample product no longer contains variability and can be checked with single system techniques. However, such an approach creates a high effort if the quality of the entire product line has to be ensured, since all possible products of the product line would have to be derived as sample product. To avoid such a high effort, the adaptation of

[2] For modeling variability in domain artifacts, different models can be used such as feature models [11] or the orthogonal variability model [22]. These and other documentation forms for variability can be transformed into a Boolean expression [1, 14]. In order to keep our approach independent from a particular variability modeling approach, we use a variability model which consists of Boolean variables and Boolean expressions.

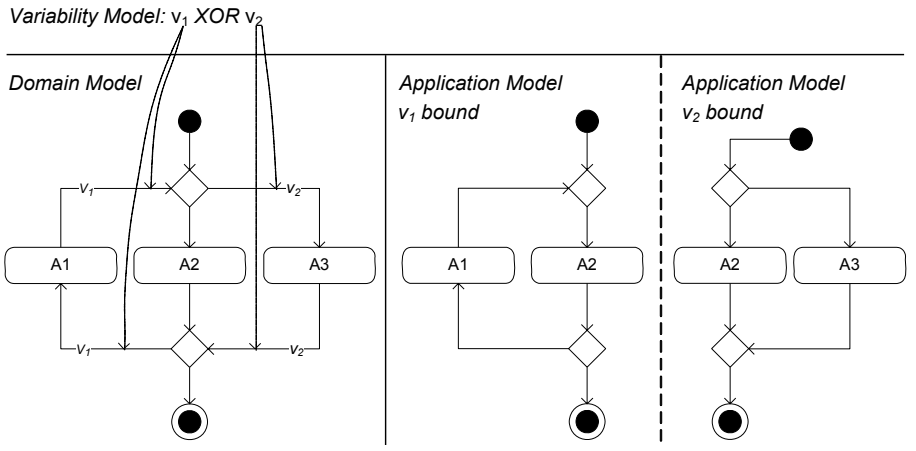


Fig. 1. Examples of domain and variability models

single system techniques or the development of new techniques is a necessary step to support comprehensive quality assurance in domain engineering [13].

A prerequisite for the adaption of techniques or development of new techniques based on activity diagrams is a precise understanding of the syntax and semantics of modeling variability in activity diagrams. The contribution of this paper is twofold.

- We discuss the related work regarding the formal definition of variability in activity diagrams in Section 2 and introduce a formal syntax and semantics for documenting variability in activity diagrams in Sections 3 and 4.
- We illustrate the applicability of our formalization approach by two use cases in Section 5 and show that our definition of variability in activity diagrams is suitable for developing quality assurance techniques that are applicable to activity diagrams containing variability.

We close our paper with a summary and outlook in Section 6.

2 Related Work

In this section, we discuss the work related to the formal definition of variability in activity diagrams. The first part of this section reflects work on the definition of a formal syntax and semantics for activity diagrams, since the understanding of the syntax and semantics of activity diagrams is a prerequisite for defining variability in activity diagrams. In the second part, we review current work on formal definition of variability in activity diagrams.

2.1 Formal Syntax and Semantics of Activity Diagrams

Formal syntax and semantics for activity diagrams (including tool support) was already discussed in several papers [6, 7, 8, 16, 31]. However, the work of Eshuis

et al. [6, 7, 8] is based on the outdated UML 1 that defined the semantics of activity diagrams based on state machines. UML 2 defines the semantics of activity diagrams as a token-based control flow structure that allows concurrent executions [19].

Linzhang et al. [16] propose an approach for the automatic derivation of test cases based on a grey-box method. The test cases are derived from UML 2 activity diagrams with a reduced set of model elements. To enable the automatic derivation of test cases, the authors formalize the syntax and semantics of the activity diagram based on Petri-nets (cf. [21]), but do not map the model elements to Petri-net elements. Störrle [28, 29, 30] as well as Störrle and Hausmann [31] focus on the formalization of UML 2 activity diagrams based on Petri-nets. They aim at formalizing all elements that were defined in the UML specification [19], in contrast to Linzhang et al. who formalized only a subset. Finally, they show that the formalization of the basic modeling elements of activity diagrams (e.g., activities and decisions) based on a Petri-net semantics is straight-forward, but modeling constructs like exceptions, streaming, etc. requires further work towards formalization.

2.2 Syntax and Semantics for Modeling Variability in Activity Diagrams

Variability in activity diagrams was discussed in recent work (e.g., [4, 10, 20, 25, 26]). The definition of variability in activity diagrams can be split into two groups. The first group of approaches uses UML stereotypes to extend the expressiveness of current modeling elements. Secondly, variability can be added to the model (i.e., activity diagrams) by modifying the UML metamodel and adding new elements, or by utilizing informal UML annotations.

Defining Variability using Stereotypes

Kamsties et al. [10] and Reuys et al. [25, 26] added a variability node to the syntax of activity diagrams using stereotypes. This node represents a variation point from the variability model. The following control flow represents the different variants that can be chosen at the specific variation point.

Robak et al. [27] also use the built-in mechanism of stereotypes in UML to model variability of activities in the UML activity diagram. The authors use decisions at decision nodes to elaborate the existence of specific variants. They do not define a formal semantics and binding rules, i.e., the approach is not automatable with tools, since product derivations based on the stereotyped activities might lead to invalid models with dangling edges.

The approaches presented in this subsection use predefined methods (stereotypes) in UML and do not extend the syntax of the language. However, they do not define the semantics of their stereotypes formally and thus an automated approach is not possible, which would include, for example, binding rules for deriving application activity diagrams, i.e. activity diagrams for specific applications of the product line without variability.

Defining Variability based on Annotations and Extensions
The goal of Braganca and Machaod [4, 20] is to support model-driven approaches for product lines. Therefore, they use activity diagrams as part of use case specifications and extend them to cover variability in use case models. They modify the current UML 2.0 use case model by adding extensions to describe different ways of branching in activity diagrams caused by variability. The semantics of their extensions is defined in natural language, but not formally. Another drawback of this approach is the extension of activity diagrams with new elements, for example, so called *ExtensionFragments* that are used to model variability in use case diagrams and are shown as *extend* relationships with an annotation. However, this implies the need for training the developers to be able to use this approach.

Hartmann et al. [9] use activity diagrams for modeling the domain test model. In their approach, they use UML annotations to model the variability in activity diagrams. The content of annotations is not defined in UML and it might include any natural language. Hence, their proposed extension has neither a formal syntax nor a formal semantics.

2.3 Conclusion of Related Work

The OMG (Object Management Group) defines the semantics of UML activity diagrams based on a token-based control flow and recent publications show that this control flow can be successfully formalized on the basis of Petri-nets [28, 29, 30, 31].

The analysis of the related work on formalizing variability in activity diagrams has shown that current approaches use semi-formal techniques like UML stereotypes or UML extensions based on natural language, which hinders a fully automated support in domain engineering. None of the existing approaches takes advantage of formalizing the variability in activity diagrams based on Petri-nets as suggested in approaches from single system engineering.

3 Foundation: Formal Definition of Activity Diagrams Based on Petri-nets

According to the definition of activity diagrams by the Object Management Group (OMG) in [19], the semantics of activity diagrams is defined as a set of all valid token flows of a system. Token flow means that the execution of one node affects and is affected by the execution of other nodes. These execution dependencies are modeled as edges in the activity diagram. A node can carry tokens that contain an object, a datum, or a focus of control [19]. If the input tokens of a node fulfill a specific condition, the node may execute. At the beginning of a certain execution, at least one token from the input edges is accepted and a token is placed on the current node. After the execution, the token is removed from the current node and passed to some or all of its output edges [19].

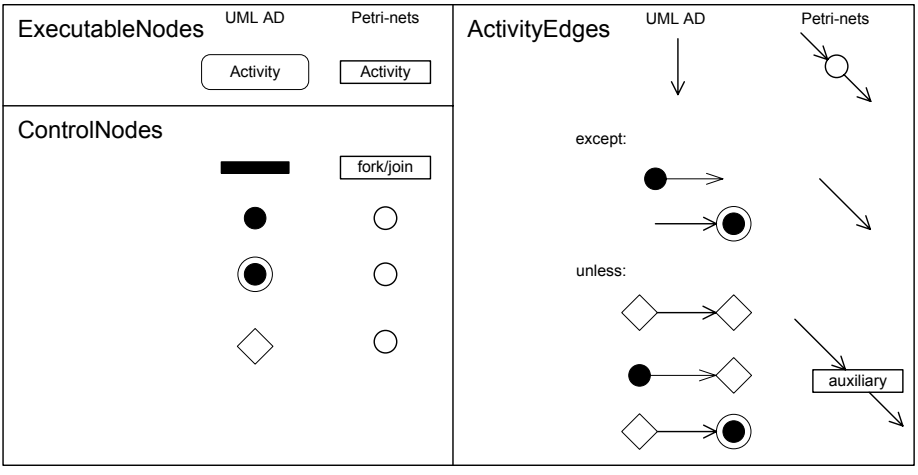


Fig. 2. Mapping activity diagram elements to Petri-net elements [31]

In [31], Störrle uses Petri-nets to formalize the OMG's semantics definition of UML activity diagrams. This provides the possibility to apply recent techniques for Petri-net analysis to activity diagrams (cf. e.g., [18]). He defines activity diagrams as a named graph structure based on a 3-tuple $\langle Name, Nodes, Edges\rangle$, where *Name* is the name of the activity diagram, *Nodes* represents the ActivityNodes, and *Edges* represents the ActivityEdges in the activity diagram [29]. According to [29], *Nodes* is defined as the 5-tuple $\langle EN, iN, fN, BN, CN\rangle$ with:

EN	the set of ExecutableNodes (i.e., elementary actions)
iN	the InitialNodes (of which there may be only one)
fN	the FinalNodes (of which there may be only one)
BN	the set of branch nodes, including MergeNodes and DecisionNodes
CN	the set of concurrency nodes, including ForkNodes, JoinNodes, and ForkJoinNodes

The set of *Edges* comprises all ActivityEdges between these nodes, except those dealing with data flow. The target Petri-net is defined as the tuple $\langle P, T, A, \underline{m}, \underline{\underline{m}}\rangle$, where the set *P* represents the places, the set *T* represents the transitions, the set *A* the arcs between the places and transitions, $\underline{m} \in P^*$ the initial, and $\underline{\underline{m}} \in P^*$ the final marking of the Petri-net, where P^* is a multiset over *P*. The translation between activity diagrams and Petri-nets is defined in [29] as follows:

$[\![\langle Nodes, Edges\rangle]\!] = \langle P, T, A, \underline{m}, \underline{\underline{m}}\rangle$, where

$$P = \{iN, fN\} \cup BN \cup \{p_a \mid a \in Edges,\ \{a_1, a_2\} \cap (EN \cup CN) \neq \varnothing$$

$$T = EN \cup CN \cup \{t_a \mid a \in Edges,\ \{a_1, a_2\} \subseteq BN \cup \{iN, fN\}\}$$

$$A = \{\langle x_{\langle from,\ to\rangle}, to\rangle, \langle from, x_{\langle from,\ to\rangle}\rangle \mid \langle from, to\rangle \in Edges\}$$

$$\underline{m} = iN$$

$$\underline{\underline{m}} = fN$$

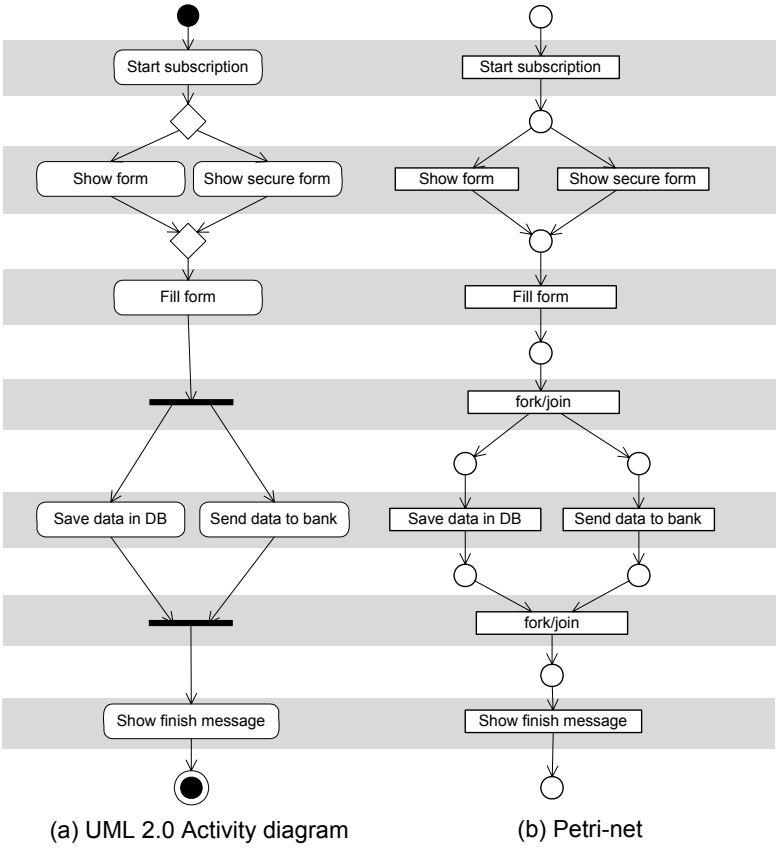


Fig. 3. Exemplary mapping of an activity diagram to a Petri-net

The variable x covers all elements in $X = P \cup T$, i.e., any Petri-net node. The variables a_1 and a_2 are shorthand for the first and the second element of a pair. ActivityEdges are used as indexes for names of places that represent the former ActivityEdge and are simplifying the definition.

In the Petri-net, places represent the former initial and final nodes and all branch nodes of the activity diagram. Additionally, every ActivityEdge is represented by a place, except for the ActivityEdges that connect any ExecutableNode or concurrency node. Every ExecutableNode and concurrency node is mapped to a transition in the Petri-net. Additionally, auxiliary transitions are added for edges connecting the initial or final node and branch nodes (see Fig. 2). The set of arcs in the Petri-net is defined as all arcs that connect the source of the ActivityEdge with the corresponding node in the Petri-net.

The described mapping is summarized in Fig. 2. A complete description of this mapping can be found in [29]. We illustrate the mapping between activity diagrams and Petri-nets with an example in Fig. 3. Fig. 3 (a) shows an activity diagram with decisions and concurrency and Fig. 3 (b) shows the corresponding Petri-net.

4 Formal Definition of Variability in Activity Diagrams

For the formal definition of variability in activity diagrams, we follow the approach of Lauenroth and Pohl [13] for the formal definition of variability in a development artifact, which consists of the following steps:

1) A formal definition of the relationship between the variability model and elements of the development artifacts.
2) A formal definition of the semantics of variability in the development artifact.
3) A formal definition of the derivation or binding process of the variability in the development artifact.

4.1 Relationship between Variability Model and Activity Diagram

The discussion of the related work on variability in activity diagrams in Section 2.2 showed that variability in an activity diagram aims at changing the flow of control in the activity diagram by defining edges as variable (see Fig. 1).

For a formal definition of this relationship, Lauenroth and Pohl [13] use the variability relationship $VRel \subseteq V \times \mathfrak{P}(D)$, where V represents the set of variants (or features) of the variability model, $\mathfrak{P}(D)$ denotes the powerset of D, and D represents the set of elements of the related development artifact. Since a variant can be related to more than one development artifact, the relationship is defined for the powerset of development artifacts. For the definition of variability in activity diagrams, the variability relationship has to be defined between the set of variants and the set of edges (see Section 3) of the activity diagram:

$$VRel_{AD} \subseteq V \times \mathfrak{P}(Edges)$$

Each edge in the activity diagram that is related to a variant is considered as *variable edge*, i.e., such an edge is selectable for a derived product of the product line. Each edge that is *not* related to a variant is considered as a *common edge* for now, i.e., such an edge is part of every derived product.

4.2 Semantics of Variability in Activity Diagrams

Based on the Petri-net semantics of activity diagrams presented in Section 3, we will provide a formal definition of the semantics for modeling variability in activity diagrams by defining variability in Petri-nets and by extending the mapping between activity diagrams and Petri-nets.

In order to achieve variability in the token flow of a Petri-net, the variability relationship for Petri-nets has to be defined between the set V of variants and the set A of arcs as follows:

$$VRel_{PN} \subseteq V \times \mathfrak{P}(A)$$

The mapping from variability in activity diagrams defined in Section 4.1 to variability in Petri-nets is defined as follows based on the definition of arcs in [28]:

$$VRel_{PN} = \{\langle v, \langle x_{\langle from,\ to\rangle}, to\rangle\rangle \mid \langle from, to\rangle \in Edges \ \wedge \ \langle v, \langle from, to\rangle\rangle \in VRel_{AD}\} \\ \cup \{\langle v, \langle from,\ x_{\langle from,\ to\rangle}\rangle\rangle \mid \langle from, to\rangle \in Edges \ \wedge \ \langle v, \langle from, to\rangle\rangle \in VRel_{AD}\}$$

This definition maps all relations between edges and variants in the variability relation of the activity diagram to the corresponding pairs of arcs and variants in the Petri-net variability relation. The mapping from edges in the activity diagram to Petri-nets can result in adding new transitions or places in the Petri-net (see Fig. 2). However, these new nodes can be neglected, because the incoming and the outgoing arcs are defined as variable by the upper mapping and thus the new nodes (e.g., the transitions) are variable.

4.3 Binding Variability in Activity Diagrams

The definition of the binding process for the variability is the third and last step for the formal definition of variability in development artifacts. The binding of variants is based on a valid selection of variants. Valid in this case means that the selection of variants (or features) obeys the constraints defined in the variability model. A valid selection of variants is denoted as follows:

$$\underline{S} = (v_1, \cdots, v_{|V|}) \in (true, false)^{|V|}$$

We define the binding approach for variability in activity diagrams based on the Petri-net semantics defined in Section 4.2. The binding approach consists of two steps:

1) All variable arcs that are not related to a selected variant are removed from the Petri-net. Assuming a valid variant selection $\underline{S}$, the set of arcs is defined as follows:

$$A' = \{\langle x_1, x_2\rangle \mid \exists \langle x_1, x_2\rangle \in A \wedge \\ (\nexists \langle v, \langle x_1, x_2\rangle\rangle \in VRel_{PN} \vee \exists \langle v, \langle x_1, x_2\rangle\rangle \in VRel_{PN}: v = true)\}$$

2) Isolated transitions and places are removed from the Petri-net. This step can, for example, be realized by a depth-first-search or breadth-first-search traversal of the Petri-net.

Now, the resulting Petri-net does not include any variability. This Petri-net can now be mapped back to a valid activity diagram based on the mapping rules presented in Section 3. We illustrate the derivation process with an example in Fig. 4. In Fig. 4 (a) the domain activity diagram is shown. It has two variable edges from the activity A1. The mapping to the corresponding Petri-net is shown in Fig. 4 (b). Now, the variability is bound and V_1 is not part of the product. The resulting Petri-net after binding the variability is shown in Fig. 4 (c). This Petri-net can now be mapped back to an activity diagram that does not include any variability (see Fig. 4 (d)). Based on the mapping semantics, the whole control flow following after the variable edge to the branch node is removed.

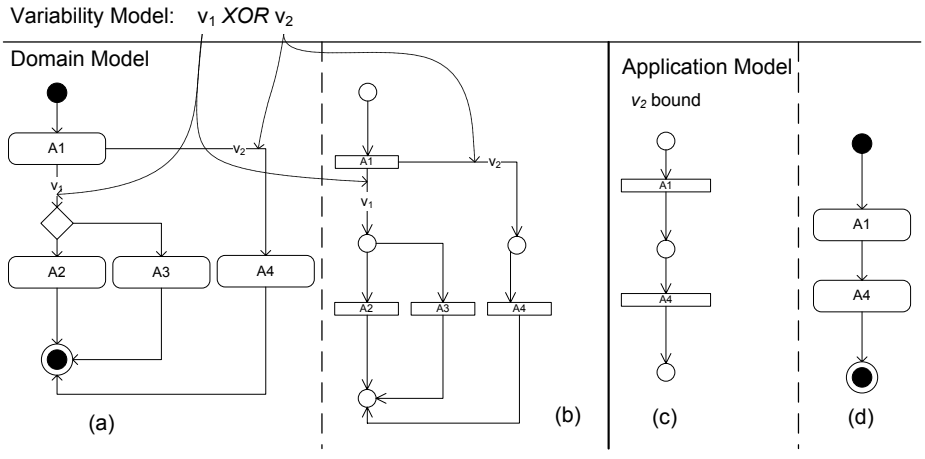


Fig. 4. Example of binding variability

5 Preliminary Evaluation

In this section, we present a preliminary evaluation of our formal definition of variability in activity diagrams. The goal of our preliminary evaluation is to show that our formal definition supports the development of useful automated quality assurances techniques for activity diagrams in domain engineering. We have selected the following use cases for the evaluation:

1) *Automated validation of variability*: This scenario deals with the validation of the defined variability in activity diagrams. A wrong definition of variability in activity diagrams may allow the derivation of invalid activity diagrams in which the final node of the activity diagram is not reachable. Based on this scenario, we present a validation algorithm that supports the product line engineers in the identification of such defects during the specification of variability in activity diagrams.
2) *Support for the sample quality assurance strategy*: The sample strategy supports quality assurance in domain engineering based on derived sample applications [17]. For this u, we present an automated approach for the selection of representative sample applications of the product line.

5.1 Validation of Variability in Activity Diagrams

Variability in activity diagrams can cause syntactic and semantic problems like isolated sub-graphs or unreachable states. Fig. 5 (a) illustrates this problem with a simplified activity diagram. If the variability in the activity diagram shown in Fig. 5 (a) is neglected, the activity diagram is valid. However, the variability model states that v_1 and v_2 cannot be selected at the same time in the same product, i.e., the edges labeled *a* and *b* will never become part of the same product and therefore it is

impossible to reach the final state in the activity diagram in any product of this product line.

Algorithm 1 Detecting invalid activity diagrams

(1) *foreach* $P \in TC$ *do*
(1.1) $V_{Sel} = \emptyset$
(1.2) *foreach* $x \in P$ *do*
(1.2.1) *If* $\exists \langle x, v \rangle \in VRel_{AD}$: $v \in V$ *then*
(1.2.1.1) $V_{Sel} = V_{Sel} \cup \{v\}$
(1.3) *If SAT-VM*(V_{Sel}) = *false then*
(1.3.1) $TC = TC \setminus P$

This problem can be detected using the detection algorithm in Algorithm 1, which works as follows. The algorithm derives all paths through the domain activity diagram by using the node reduction algorithm proposed by Beizer [2]. The result of the node reduction algorithm is a formula representing all paths through the activity diagram. In Fig. 5 (a) for example, the path expression is *abcdg+abefg*. This can be represented by the set $TC = \{(a, b, c, d, g); (a, b, e, f, g)\}$, since now variable edges are ignored. In the first step, the derived paths are checked by iterating over the set *TC*. The current path is iterated and checked against the variability model. If the current edge x of the path has a variant relation (step 1.2.1), i.e., $\exists \langle x, v \rangle \in VRel_{AD}$, the variant v has to be added to the set V_{Sel} (step 1.2.1.1). In the example above, the first sequence is *abcdg*. The edges *a* and *b* are related to v_1 and v_2, i.e., after iteration over the first path expression, the set V_{Sel} consists of the elements $V_{Sel} = \{v_1, v_2\}$. In step 1.3, the variant selection v_1 and v_2 is checked using the variability model whether it is a valid variant selection, e.g., by using a SAT solver (function *SAT-VM*). The SAT solver function returns false, i.e., the variant selection is invalid. Because it is invalid, the path *abcdg* is removed from *TC* and *TC* now consists of $\{(a, b, e, f, g)\}$ (step 1.3.1). The next element of *TC* is validated in the same way. Since it is also not a valid expression, it is removed and the set *TC* is empty, i.e., $TC = \emptyset$.

The empty set means that it is not possible to make a valid variant selection that leads to a valid path in the variable activity diagram. Thus, the specified activity diagram is not valid.

5.2 Identification of Sample Products for Product Line Quality Assurance

The sample strategy is a popular approach for quality assurance in domain engineering and works as follows: based on the domain artifacts of the product line, one or more sample products are derived and tested [17]. The main advantage of this strategy is that quality assurance techniques from single system engineering can be applied to the derived sample products. However, the quality of the results of the sample strategy mainly depends on the selected sample products. A set of sample products that is not representative for the product line only allows a limited conclusion related to the quality of the entire product line. Therefore, the main challenge of the sample strategy is the identification of representative products.

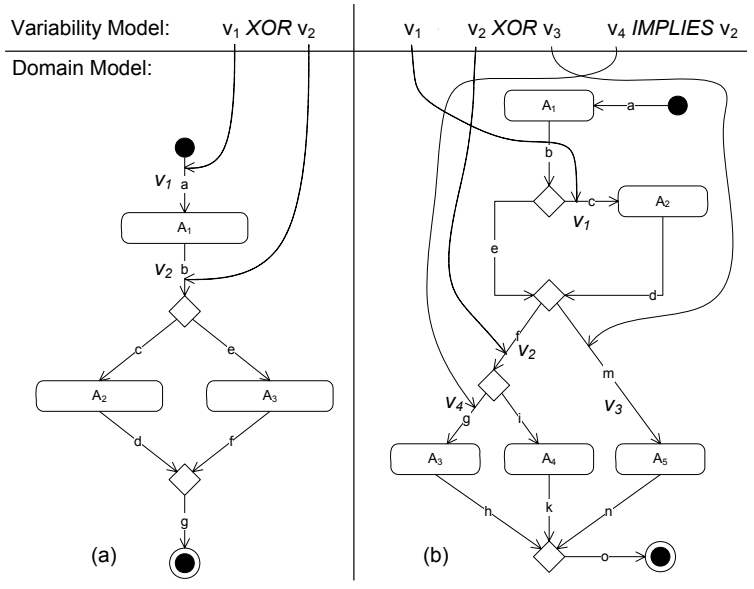


Fig. 5. Sample test models

In the following, we present an automated selection algorithm for the identification of sample products that takes advantage of our formal definition of variability in activity diagrams. The algorithm consists of two steps. First, a formula for the number of test cases based on the domain activity diagram is derived. This formula includes variables representing the variants. In the second step, this formula has to be optimized to return the highest number of test cases regarding a valid variant selection.

The calculation of the number of paths through the domain test model is again based on the node reduction algorithm proposed by Beizer [2]. This algorithm implicates the use of the path coverage criterion. However, the path coverage criterion can uncover most of the errors in software (cf. [15]). Loops in test models have to be handled separately, because the number of test case scenarios depends on the strategy for loop handling. To illustrate the approach, we use the example provided in Fig. 5 (b).

In the first step, we have to derive the formula for the number of paths using the domain test model. Therefore, the node reduction algorithm proposed by Beizer [2] is utilized. Based on an adjacency matrix, all common edges are replaced by *1*, whereas all variable edges are replaced by the related variant (e.g., *c* is replaced by v_1). Then, the node reduction algorithm is executed. The result of the algorithm is a formula. For the example in Fig. 5 (b), the formula is:

$$v_1+v_1v_3+v_1v_4+v_1v_3v_4+v_2+v_2v_4 = v_1(1+v_3+v_4+v_3v_4)+v_2(1+v_4)$$

Applying a valid variant selection, the variables can be replaced by *0* (if the variant is not part of a product) or *1* (if the variant is part of a product) and the number of test cases can be calculated. For example, if the variants *v1* and *v3* are chosen, this product can be tested with *two* test cases (*abemno* and *abcdmno*). In the second step,

the maximum of this formula has to be calculated regarding the variability model. Obviously, the highest number of test cases results from the selection of each variant. However, the variability model has to be regarded by the variant selection. This optimization can be solved, for example, by a greedy algorithm (cf. [5]) to find the optimal combination of variants.

6 Summary and Outlook

In this paper, we presented a formal definition of the syntax and semantics of variability in activity diagrams based on Petri-net semantics. The use of Petri-nets lends itself to the formal definition of the syntax and semantics of variability in activity diagrams, since the OMG defines the semantics of activity diagrams on the basis of token-flows [19] and the state-of-the-art already provides a profound definition of the syntax and semantics of activity diagrams based on Petri-nets [29].

The formal definition of the syntax and semantics of a modeling language is a prerequisite for an automated analysis approach. Therefore, our definition provides the theoretical foundation for a comprehensive support of quality assurance in domain engineering.

We showed the applicability of our definition in two different use cases. The first use case presented an automated validation algorithm for variability in activity diagrams. The second use case presented an automated approach for the selection of representative sample products in domain engineering that can be used in the sample quality assurance strategy.

In our future research, we will follow two directions. First, we plan to implement our definitions in a tool prototype that can be used in case studies with our industrial partners. Second, we plan to investigate on further possibilities for the automated support of quality assurance in domain engineering based on activity diagrams.

References

1. Batory, D.S.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
2. Beizer, B.: Software Testing Techniques, 2nd edn. Van Nostrand Reinhold, New York (1990)
3. Binder, R.: Testing Object-Oriented Systems – Models, Patterns, and Tools. Addison-Wesley, Reading (1999)
4. Braganca, A., Machado, R.J.: Extending UML 2.0 Metamodel for Complementary Usages of the ≪extend≫ Relationship within Use Case Variability Specification. In: Proceedings of the 10th International Conference on Software Product Lines, SPLC 2006, pp. 123–130. IEEE Computer Society, Los Alamitos (2006)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009) ISBN 978-0262533058

6. Eshuis, H.: Semantics and Verification of UML Activity Diagrams for Workflow Modelling. PhD thesis. Univ. of Twente. CTIT Ph.D. thesis series No. 02-44 (2002) ISBN 9036518202
7. Eshuis, R.: Symbolic model checking of UML activity diagrams. ACM Trans. Softw. Eng. Methodol. 15(1), 1–38 (2006)
8. Eshuis, R., Wieringa, R.: Tool Support for Verifying UML Activity Diagrams. IEEE Transactions on Software Engineering 30(7), 437–447 (2004)
9. Hartmann, J., Vieira, M., Ruder, A.: A UML-based Approach for Validating Product Lines. In: Geppert, B., Krueger, C., Jenny Li, J. (eds.) Proceedings of the International Workshop on Software Product Line Testing (SPLiT) 2004. Boston, USA. Avaya labs ALR-2004-031 (2004)
10. Kamsties, E., Pohl, K., Reis, S., Reuys, A.: Testing Variabilities in Use Case Models. In: Proceedings of the 5th International Workshop on Software Product-Family Engineering (2003)
11. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study Software Engineering Institute. Carnegie Mellon University, Pittsburgh (1990)
12. Larsen, K., Nyman, U., Wąsowski, A.: Modal I/O Automata for Interface and Product Line Theories. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 64–79. Springer, Heidelberg (2007)
13. Lauenroth, K., Pohl, K.: Towards Automated Consistency Checks of Product Line Requirements Specifications. In: Stirewalt, K., Egyed, A., Fischer, B. (eds.) Proceedings of the 27th International Conference on Automated Software Engineering (ASE 2007), Atlanta, USA, November 5-9 (2007)
14. Lauenroth, K., Töhning, S., Pohl, K.: Model Checking of Domain Artifacts in Product Line Engineering. In: Proceedings of the 24th International Conference on Automated Software Engineering (ASE), New Zealand, pp. 373–376 (2009)
15. Liggesmeyer, P.: Software Qualität: Testen, Analysieren und Verifizieren von Software (in german), 2nd edn. Spektrum Akademischer Verlag (2009)
16. Linzhang, W., Jiesong, Y., Xiaofeng, Y., Jun, H., Xuandong, L., Guoliang, Z.: Generating Test Cases from UML Activity Diagram based on Gray-Box Method. In: Proceedings of the 11th Asia-Pacific Software Engineering Conference, pp. 284–291. IEEE Computer Society, Washington (2004)
17. Metzger, A.: Quality Issues in Software Product Lines: Feature Interactions and Beyond. In: du Bousquet, L., Richier, J.-L. (eds.) Feature Interactions in Software and Communication Systems IX, International Conference on Feature Interactions in Software and Communication Systems, ICFI 2007, Grenoble, France. IOS Press, Amsterdam (2007)
18. Murata, T.: Petri-nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
19. Object Management Group: UML 2.2 Superstructure and Infrastructure, http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML
20. Olimpiew, E.M., Gomaa, H.: Model-based Test Design for Software Product Lines. In: Thiel, S., Pohl, K. (eds.) Software Product Lines, Proceedings of 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, Second Volume (Workshops), Lero Int. Science Centre, University of Limerick, Ireland (2008)
21. Petri, C.A.: Kommunikation mit Automaten. In: Schriften des Rheinisch-Westfälischen Institutes für instrumentelle Mathematik an der Universität Bonn, Bonn (1962)
22. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering – Foundations, Principles, Techniques. Springer, Heidelberg (2005)

23. Pohl, K., Metzger, A.: Software Product Line Testing – Exploring Principles and Potential Solutions. Communications of the ACM 49(12), 78–81 (2009)
24. Reis, S., Metzger, A., Pohl, K.: Integration Testing in Software Product Line Engineering: A Model-Based Technique. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 321–335. Springer, Heidelberg (2007)
25. Reuys, A., Kamsties, E., Pohl, K., Reis, S.: Model-Based System Testing of Software Product Families. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 519–534. Springer, Heidelberg (2005)
26. Reuys, A., Reis, S., Kamsties, E., Pohl, K.: The ScenTED Method for Testing Software Product Lines. In: Käkölä, T., Duenas, J.C. (eds.) Software Product Lines – Research Issues in Engineering and Management, pp. 479–520. Springer, Heidelberg (2006)
27. Robak, D., Franczyk, B., Politowicz, K.: Extending the UML for Modeling Variability for System Families. International Journal of Applied Mathematics and Computer Science 12(2), 285–298 (2002)
28. Störrle, H.: Semantics and Verification of Data Flow in UML 2.0 Activities. In: Minas, M. (ed.) Proceedings of the Workshop on Visual Languages and Formal Methods (VLFM 2004). Electronic Notes in Theoretical Computer Science, vol. 127(4), pp. 35–52. Elsevier, Amsterdam (2004)
29. Störrle, H.: Semantics of Control-Flow in UML 2.0 Activities. In: Bottoni, P., Hundhausen, C., Levialdi, S., Tortora, G. (eds.) Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 235–242. Springer, Heidelberg (2004)
30. Störrle, H.: Structured Nodes in UML 2.0 Activities. Nordic Journal of Computing 11(3), 279–302 (2004)
31. Störrle, H., Hausmann, J.H.: Towards a formal semantics of UML 2.0 activities. In: Liggesmeyer, P., Pohl, K., Goedicke, M. (eds.) Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik. Lecture Notes in Informatics, pp. 117–128. Gesellschaft für Informatik (2005) ISBN 3-88579-393-8

Delta-Oriented Programming of Software Product Lines

Ina Schaefer[1], Lorenzo Bettini[2], Viviana Bono[2],
Ferruccio Damiani[2], and Nico Tanzarella[2]

[1] Chalmers University of Technology, 421 96 Gothenburg, Sweden
schaefer@chalmers.se
[2] Dipartimento di Informatica, Università di Torino, C.so Svizzera, 185 - 10149 Torino, Italy
{bettini,bono,damiani}@di.unito.it, nicotanz@libero.it

Abstract. Feature-oriented programming (FOP) implements software product lines by composition of feature modules. It relies on the principles of stepwise development. Feature modules are intended to refer to exactly one product feature and can only extend existing implementations. To provide more flexibility for implementing software product lines, we propose delta-oriented programming (DOP) as a novel programming language approach. A product line is represented by a core module and a set of delta modules. The core module provides an implementation of a valid product that can be developed with well-established single application engineering techniques. Delta modules specify changes to be applied to the core module to implement further products by adding, modifying and removing code. Application conditions attached to delta modules allow handling combinations of features explicitly. A product implementation for a particular feature configuration is generated by applying incrementally all delta modules with valid application condition to the core module. In order to evaluate the potential of DOP, we compare it to FOP, both conceptually and empirically.

1 Introduction

A *software product line* (SPL) is a set of software systems with well-defined commonalities and variabilities [13,29]. The variabilities of the products can be defined in terms of *product features* [16], which can be seen as increments of product functionality [6]. *Feature-oriented programming* (FOP) [10] is a software engineering approach relying on the principles of *stepwise development* [9]. It has been used to implement SPLs by composition of *feature modules*. In order to obtain a product for a feature configuration, feature modules are composed incrementally. In the context of object-oriented programming, feature modules can introduce new classes or refine existing ones by adding fields and methods or by overriding existing methods. Feature modules cannot remove code from an implementation. Thus, the design of a SPL always starts from a base feature module which contains common parts of all products. Furthermore, a feature module is intended to represent exactly one product feature. If the selection of two optional features requires additional code for their interaction, this cannot be directly handled leading to the optional feature problem for SPLs [20].

In this paper, we propose *delta-oriented programming* (DOP) as a novel programming language approach particularly designed for implementing SPLs, based on the concept of program deltas [32,31]. The goal of DOP is to relax the restrictions of FOP

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 77–91, 2010.

and to provide an expressive and flexible programming language for SPL. In DOP, the implementation of a SPL is divided into a *core module* and a set of *delta modules*. The core module comprises a set of classes that implement a complete product for a valid feature configuration. This allows developing the core module with well-established single application engineering techniques to ensure its quality. Delta modules specify changes to be applied to the core module in order to implement other products. A delta module can add classes to a product implementation or remove classes from a product implementation. Furthermore, existing classes can be modified by changing the super class, the constructor, and by additions, removals and renamings of fields and methods. A delta module contains an application condition determining for which feature configuration the specified modifications are to be carried out. In order to generate a product implementation for a particular feature configuration, the modifications of all delta modules with valid application condition are incrementally applied to the core module. The general idea of DOP is not restricted to a particular programming language. In order to show the feasibility of the approach, we instantiate it for JAVA, introducing the programming language DELTAJAVA.

DOP is a programming language approach especially targeted at implementing SPLs. The delta module language includes modification operations capable to remove code such that a flexible product line design and modular product line evolution is supported. The application conditions attached to delta modules can be complex constraints over the product features such that combinations of features can be handled explicitly avoiding code duplication and countering the optional feature problem [20]. The ordering of delta module application can be explicitly defined in order to avoid conflicting modifications and ambiguities during product generation. Using a constraint-based type system, it can be ensured that the SPL implementation is well formed. This yields that product generation is safe, which means that all resulting products are type correct. In order to evaluate the potential of DOP, we compare it with FOP, both on a conceptual and on an empirical level using case examples studied for FOP.

2 Delta-Oriented Programming

In order to illustrate delta-oriented programming in DELTAJAVA, we use the *expression product line* (EPL) as described in [25] as running example. We consider the following grammar for expressions:

Exp ::= Lit | Add | Neg Lit ::= <non−negative integers> Add ::= Exp "+" Exp Neg ::= "−" Exp

Two different operations can be performed on the expressions described by this grammar: first, printing, which returns the expression as a string, and second, evaluation, which computes the value of the expression. The set of products in the EPL can be described with a feature model [16], see Figure 1. It has two feature sets, the ones concerned with data Lit, Add, Neg and the ones concerned with operations Print and Eval. Lit and Print are mandatory features. The features Add, Neg and Eval are optional.

Core Module. In DELTAJAVA, a software product line is implemented by a core module and a set of delta modules. A *core module* corresponds to the implementation of a product for a valid feature configuration. It defines the starting point for generating all

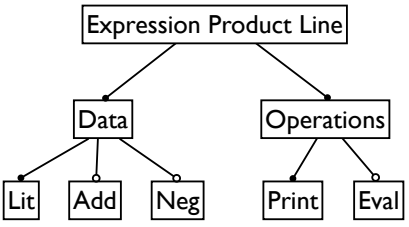


Fig. 1. Feature Model for Expression Problem Product Line

```
core Print, Lit {
  interface Exp { void print(); }

  class Lit implements Exp {
        int value;
        Lit(int n) { value=n; }
        void print() { System.out.print(value); }
    }
}
```

Listing 1. Core module implementing Lit and Print features

other products by delta module application. The core module depends on the underlying programming language used to implement the products. In the context of this work, the core module contains a set of JAVA classes and interfaces. These are enclosed in a `core` block additionally specifying the implemented features by listing their names:

core <Feature names> { <Java classes and interfaces> }

The product represented by the core module can be any valid product. Thus, it has to implement at least the mandatory features and a minimal set of required alternative features, if applicable. Note that the core module is not uniquely determined, as illustrated in Section 3. Choosing the core module to be a valid product allows to develop it with standard single application engineering techniques to ensure its quality and to validate and verify it thoroughly with existing techniques. Listing 1 contains a core module for the EPL. It implements the features Lit and Print.

Delta Modules. Delta modules specify changes to the core module in order to implement other products. The alterations inside a delta module act on the class level (by adding, removing and modifying classes) and on the class structure level (by modifying the internal structure of a class by changing the super class or the constructor, or by adding, removing and renaming fields and methods).[1] Furthermore, a delta module can add, remove or modify interfaces by adding, removing or renaming method signatures. An application condition is attached to every delta module in its when clause determining for which feature configurations the specified alterations are to be carried out. Application conditions are propositional constraints over features. This allows specifying delta modules for combinations of features or to explicitly handle the absence of

[1] Renaming a method does not change calls to this method. The same holds for field renaming.

```
delta DEval when Eval {
    modifies interface Exp { adds int eval(); }
    modifies class Lit {
        adds int eval() { return value; }
    }
}
```

```
delta DAdd when Add {
    adds class Add implements Exp {
        Exp expr1;  Exp expr2;
        Add(Exp a, Exp b) { expr1=a; expr2=b; }
    }
}
```

Listing 2. Delta modules for Eval and Add features

features. The number of delta modules required to implement all products of a product line depends on the granularity of the application conditions. In general, a delta module has the following shape:

```
delta <name> [after <delta names>] when <application condition> {
    removes <class or interface name>
    adds class <name> <standard Java class>
    adds interface <name> <standard Java interface>
    modifies interface <name> { <remove, add, rename method header clauses> }
    modifies class <name> { <remove, add, rename field clauses> <remove, add, rename method clauses> }
}
```

The left of Listing 2 shows the delta module corresponding to the Eval feature of the EPL. It modifies the interface Exp by adding the method signature for the eval method. Furthermore, it modifies the class Lit by adding the method eval. The when clause denotes that this delta module is applied for every feature configuration in which the Eval feature is present. The right of Listing 2 shows the delta module required for the Add feature adding the class Add. The when clause specifies that the delta module is applied if the features Add is present.

Product Generation. In order to obtain a product for a particular feature configuration, the alterations specified by delta modules with valid application conditions are applied incrementally to the core module. The changes specified in the same delta module are applied simultaneously. In order to ensure, for instance, that a class to be modified exists or that a modification of the same method by different delta modules does not cause a conflict, an ordering on the application of delta modules can be defined by means of the after clause. This ordering implies that a delta module is only applied to the core module after all delta modules with a valid application condition mentioned in the after clause have been applied. The partial ordering on the set of delta modules defined by the after clauses captures the necessary dependencies between the delta modules, which are usually semantic requires relations. As an example, consider the left of Listing 3 which depicts the delta module introducing the evaluation functionality for the addition expression. Its when clause specifies that it is applied, if both the Add and the Eval features are present. In the after clause, it is specified that this delta module has to be applied after the delta module DAdd because the existence of the class Add is assumed. The implementation of the delta module DAddPrint is similar. Note that specifying that a delta module A has to be applied after the delta module B does *not* mean that A requires B: it only denotes that if a feature configuration satisfies the when clause of both A and B, then B must be applied before A.

The generation of a product for a given feature configuration consists of the following steps, performed automatically by the system: (i) Find all delta modules with a

```
delta DAddEval after DAdd when Add && Eval {
    modifies class Add {
        adds int eval()
            { return expr1.eval() + expr2.eval(); }
    }
}
```

```
delta DAddPrint after DAdd when Add && Print {
    modifies class Add {
        adds void print()
            { expr1.print();
              System.out.print(" + "); expr2.print(); }
    }
}
```

Listing 3. Delta modules for Add and Eval and for Add and Print features

```
interface Exp { void print(); int eval(); }

class Lit implements Exp {
    int value;
        Lit(int n) { value=n; }
        void print()
            { System.out.print(value); }
        int eval()
            { return value; }
}
```

```
class Add implements Exp {
    Exp expr1;  Exp expr2;
    Add(Exp a, Exp b) { expr1=a; expr2=b;}
    void print()
        { expr1.print();
          System.out.print(" + "); expr2.print(); }
    int eval()
        { return expr1.eval() + expr2.eval();}
}
```

Listing 4. Generated implementation for Lit, Add, Print, Eval features

valid application condition according to the feature configuration (specified in the when clause); and (*ii*) Apply the selected delta modules to the core module in any linear ordering respecting the partial order induced by the after clauses.

As an example of a product implementation generated by delta application, consider Listing 4 which shows the implementation of the Lit, Add, Print, Eval features of the EPL. It is an ordinary JAVA program containing the interface Exp and the classes Lit and Add. The implementation is obtained by applying the delta modules depicted in Listings 2 and 3 to the core module (cf. Listing 1) in any order in which DAddEval and DAddPrint are applied after DAdd.

Safe Program Generation. The automatic generation of products by delta application is only performed if the DELTAJAVA product line implementation is well-formed. Well-formedness of a product line means that all delta modules associated to a valid feature configuration are well-formed themselves and applicable to the core module in any order compatible with the partial order provided by the after clauses. The partial order ensures that all compatible application orders generate the same product. A delta module is well-formed, if the added and removed classes are disjoint and if the modifications inside a class target disjoint fields and methods. A delta module is applicable to a product if all the classes to be removed or modified exist, all methods and fields to be removed or renamed exist and if classes, methods and fields to be added do not exist. Furthermore, all delta modules applicable for the same valid feature configuration that are not comparable with respect to the after partial order must be compatible. This means that no class is added or removed in more than one delta module, and for every class modified in more than one delta module the fields or methods added, modified and renamed are disjoint. This implies that all conflicts between modifications targeting the same class have to be resolved by the ordering specified with the after clauses.

```
features Lit, Add, Neg, Print, Eval
configurations Lit && Print && (Add | Neg | Eval )
core Lit, Print { [ ... ] }
delta DEval when Eval { [...] }
delta DAdd when Add { [...] }
delta DAddPrint after DAdd when Add && Print { [...] }
delta DAddEval after DAdd when Add && Eval { [...] }
delta DNeg  when Neg { [...] }
delta DNegPrint after DNeg when Neg && Print { [...] }
delta DNegEval after DNeg when Neg && Eval { [...] }
```

Listing 5. Product Line Implementation in DELTAJAVA starting from Simple Core

In order to ensure well-formedness, DELTAJAVA is accompanied by a constraint-based type system (not shown in this paper). For each delta module in isolation, a set of constraints is generated that refers to the classes, methods or fields required to exist or not to exist for the delta module to be applicable. Then, for every valid feature configuration, only by checking the constraints, it can be inferred whether delta module application will lead to a well-typed JAVA program. The separate constraint generation for each delta module avoids reinspecting all delta modules if only one delta is changed or added. Furthermore, if an error occurs, it can easily be traced back to the delta modules causing it.

3 Implementing Software Product Lines

The delta-oriented implementation of a SPL in DELTAJAVA comprises an encoding of the feature model, the core module and a set of delta modules necessary to implement all valid products. The feature model is described by its basic features and a propositional formula describing the valid feature configurations (other representations might be considered [6]). In this section, we show how SPLs are flexibly implemented in DELTAJAVA starting from different core products using the EPL as illustration.

Starting from a Simple Core. The core module of a DELTAJAVA product line contains an implementation of a valid product. One possibility is to take only the mandatory features and a minimal number of required alternative features, if applicable. In our example, the Lit and Print features are the only mandatory features. Listing 1 shows the respective core module serving as starting point of a SPL implementation starting from a simple core. In order to represent all possible products, delta modules have to be defined that modify the core product accordingly. For the EPL starting from the simple core, this are the delta modules depicted in Listings 2 and 3 together with three additional delta modules implementing the Neg feature alone as well as in combination with the Print feature and the Eval feature. Their implementation is similar to the implementation of the DAdd, DAddPrint and DAddEval delta modules and, thus, not shown here. Listing 5 shows the complete implementation of the EPL containing the encoding of the feature model (cf. Figure 1), the core module and the delta modules. For space reasons, the concrete implementations of the core and delta modules are omitted.

```
delta DRemEval when !Eval && Add {
    modifies interface Exp { removes eval;}
    modifies class Lit { removes eval;}
    modifies class Add { removes eval;}
}
```

Listing 6. Removing Eval from Complex Core

```
features Lit, Add, Neg, Print, Eval
configurations Lit && Print && (Add | Neg | Eval )
core Lit, Print, Add, Eval  { [ ... ]  }
delta DRemEval when !Eval && Add { [...] }
delta DRemAdd when !Add { [...] }
delta DRemAddEval when !Add && !Eval { [...] }
delta DNeg when Neg { [...] }
delta DNegPrint after DNeg when Neg && Print { [...] }
delta DNegEval after DNeg when Neg && Eval { [...] }
```

Listing 7. Product Line Implementation in DELTAJAVA starting from Complex Core

Starting from a Complex Core. Alternatively, a SPL implementation in DELTAJAVA can start from any product for a valid feature configuration containing a larger set of features. The advantage of the more complex core product is that all included functionality can be developed, validated and verified with standard single application techniques. In order to illustrate this idea, we choose the product with the Lit, Add, Print and Eval features as core product whose implementation is contained in Listing 4. In order to provide product implementations containing less features, functionality has to be removed from the core. Listing 6 shows a delta module that removes the evaluation functionality from the complex core. It is applied to the core module if the Eval feature is not included in a feature configuration, but the Add feature is selected. This means that only the eval method from the Add class is removed, but not the Add class itself.

Listing 7 shows the implementation of the EPL starting from the complex core module depicted in Listing 4. In addition to the delta module DRemEval, a delta module DRemAdd (not shown here) is required to remove the Add class if the Add feature is not selected, and a third delta module DRemAddEval (not shown here) is required to remove the eval method from the Lit class and the Exp interface, in case both the Eval and the Add features are not selected. Further, the delta modules DNeg, DNegPrint and DNegEval as in the previous implementation are required.

4 Comparing Delta-Oriented and Feature-Oriented Programming

In order to evaluate DOP of SPLs, we compare it with FOP [10]. Before the comparison, we briefly recall the main concepts of FOP.

4.1 Feature-Oriented Programming

In FOP [10], a program is incrementally composed from feature modules following the principles of stepwise development. A feature module can introduce new classes and

```
interface Exp { String print(); }

class Lit implements Exp {
    int value;
    Lit(int n) { value=n; }
    void print() { System.out.print(value); }
}
```

(a) LitPrint feature module

```
class Add implements Exp {
    Exp x;
    Exp y;
    Add( Exp x, Exp y ) { this.x = x; this.y = y; }
    public String print() { return x + "+" + y; }
}
```

(b) AddPrint feature module

```
refines interface Exp { int eval(); }

refines class Lit {
    public int eval() { return value; }
}
```

(c) LitEval feature module

```
refines class Add {
    public int eval()
        { return x.eval() + y.eval(); }
}
```

(d) AddEval feature module

Listing 8. Feature Modules for EPL in JAK

refine existing ones. The concept of stepwise development is introduced in GenVoca [9] and extended in AHEAD [10] for different kinds of design artifacts. For our comparison, we restrict our attention to the programming language level of AHEAD, i.e., the JAK language, a superset of JAVA containing constructs for feature module refinement.

In order to illustrate FOP in JAK, we recall the implementation of the EPL presented in [25]. The five domain features, shown in the feature model in Figure 1, are transformed into six feature modules. The difference between the number of domain features and the number of feature modules is due to the fact that combinations of domain features cannot be dealt with explicitly in JAK. Therefore, a suitable encoding of the domain features has to be chosen. This results in the feature modules for the feature combinations LitPrint, AddPrint and NegPrint combining every data feature with the Print feature. Furthermore, for each data feature, there is a feature module adding the evaluation operation, i.e., LitEval, AddEval and NegEval. The JAK code implementing the feature modules LitPrint, AddPrint, LitEval, and AddEval is shown in Listing 8. The generation of a product starts from the base feature module LitPrint. A program containing the Lit, Add, Print, and Eval features can be obtained by composing the feature modules as follows: LitPrint • AddPrint • LitEval • AddEval. The code of the resulting program is as shown in Listing 4.

4.2 Comparison

Both delta modules and features modules support the modular implementation of SPLs. However, they differ in their expressiveness, the treatment of domain features, solutions for the optional features problem, guarantees for safe composition and support for evolution. Both techniques scale to a general development approach [10,31].

Expressiveness. Feature modules can introduce new classes or refine existing ones following the principles of stepwise development [10]. The design of a SPL always starts from the base feature module containing common parts of all products. In contrast, delta modules support additions, modifications and removals of classes, methods and fields.

This allows choosing any valid feature configuration to be implemented in the core module and facilitates a flexible product line design starting from different core products, as shown in Section 3. The core module contains an implementation of a complete product. This allows developing and validating it with well-established techniques from single application engineering, or to reengineer it before starting the delta module programming to ensure its quality. For verification purposes, it might save analysis effort to start with a complex product, check the contained functionality thoroughly and remove checked functionality in order to generate other products. In JAK, an original method implementation before refinement can be accessed with a Super() call. Delta modules currently do not have an equivalent operation. However, a Super() call in delta modules could be encoded by renaming the method to be accessed and adding a corresponding call during program generation.

Domain Features. In FOP, domain-level features are intentionally separated from feature modules in order to increase the reusability of the refinements. The mapping from domain features to the feature modules is taken care of by external tools. In the AHEAD Tool Suite [10], the external tool *guidsl* [6] supports this aspect of product generation. For a given domain feature configuration, *guidsl* provides a suitable composition of the respective feature modules.

In a DELTAJAVA implementation, the features of the feature model and the corresponding constraints are explicitly specified. This allows reasoning about feature configurations within the language. For product generation, it can be established that the provided feature configuration is valid, such that only valid products are generated. For each delta module, the application condition ranges over the features in the feature model such that the connection of the modifications to the domain-level features is made explicit. This limits the reusability of delta modules for another SPL, but allows static analysis to validate the design of the very product line that is implemented. It can be checked whether the application condition of a delta module can actually evaluate to true for any valid feature configuration. Otherwise, the delta module will never be applied and can be removed. Furthermore, for a given feature configuration, the set of applicable delta modules can be determined directly without help of external tools. If, for instance, a new delta module has to be added, it is easy to learn about the consequences and potential conflicts in the existing implementation.

The Optional Feature Problem. The optional feature problem [20] occurs when two optional domain features require additional code for their interaction. Feature modules [10] are not intended to refer to combinations of features. Thus, one way to solve the optional feature problem is to move code belonging to one feature to a feature module for another feature, similar to the combination of domain features in the EPL. This solution violates the separation of concerns principle [20] and leads to a non-intuitive mapping between domain features and feature modules. In the Graph Product Line [24] implementation [5] (cf. Section 5), the optional feature problem is solved by multiple implementations per domain feature which leads to code duplications. Alternatively, derivative modules [23] can be used. In this case, a feature module is split into a base module only containing introductions and a set of derivative modules only containing

refinements that are necessary if other features are also selected. However, this may result in a large number of small modules and might not scale in practice [20].

In contrast, the optional feature problem can be solved in DOP within the language. A delta module does not correspond to one domain feature, but can refer to any combination of features. By the application condition of a delta module, the feature configurations the delta module is applied for are made explicit such that code only required for feature interaction can be clearly marked. In particular, delta modules can implement derivative modules. The implementation of the EPL in Listing 5 follows the derivative principle. Moreover, code duplication between two features modules can be avoided by factoring common code into a separate delta module that is applied if at least one of the respective features is selected.

Safe Composition. Feature composition in FOP is performed in a fixed linear order. This linear ordering has to be provided before feature module composition to avoid conflicting modifications. In DOP, the partial order specified in the after clauses of delta modules captures only essential dependencies and semantic requirements between different modifications of the same class or method. Instead of specifying a partial order, conflicting modifications between delta modules could also be prohibited completely at the price of writing additional delta modules for the respective combinations. Thus, the partial order is a compromise between modularity and a means to resolve conflicting modifications without increasing the number of delta modules.

During feature module composition, it is not guaranteed that the resulting program is correct, e.g., that each referenced field or method exists. Such errors are only raised during compilation of the generated program. Recently, there have been several approaches to guarantee safety of feature module composition [2,3,14,35] by means of external analysis or type systems. DELTAJAVA has an integrated constraint-based type system guaranteeing that the generated program for every valid feature configuration is type correct and that all conflicts are resolved by the partial order. Constraints are generated for each delta module in isolation such that an error can be traced back to the delta modules where it occurred. Additionally, changed or added delta modules do not require re-checking the unchanged delta modules.

Product Line Evolution. Product lines are long-lived software systems dealing with changing user requirements. For example, if in the EPL printing should become an optional feature, the JAK implementation has to be refactored to separate the printing functionality from the implementation of the data. In the DELTAJAVA implementation, only one delta module has to be added to remove the printing functionality from the simple as well as from the complex core, while all other delta modules remain unchanged. To this end, the expressivity of the modification operations in delta modules supports modular evolution of SPL implementations.

Scaling Delta-oriented Programming. The AHEAD methodology [10] for developing software by stepwise development is not limited to the implementation level and has been instantiated to other domain-specific languages as well as to XML. Similarly, the concepts of DOP can be applied to other programming or modeling languages. In [32],

Table 1. Summary of Comparison

	Feature-oriented Programming	**Delta-oriented Programming**
Expressiveness	Design from Base Module	Design from Any Product
Domain Features	Bijection between Features and Feature Modules	Delta Modules for Feature Combinations
Optional Feature Problem	Rearrange Code, Multiple Impl., Derivative Modules	Direct Implementation of Interaction
Safe Composition	External Tools, Type Systems	Partial Order for Conflict Resolution, Type System
Evolution	Refactoring	Addition of Delta Modules

a seamless delta-oriented model-driven development process is proposed. The variability structure in terms of core and delta modules has to be determined only once for an initial delta-oriented product line representation on a high level of abstraction. By stepwise refinement of the delta-oriented product models without changing the variability structure, a DELTAJAVA implementation of a SPL can eventually be obtained. In this way, product variability can be managed in the same manner on all levels during SPL development.

Summary. FOP is a general software engineering approach based on the principles of stepwise development that has been used to implement SPLs. In contrast, DOP is specifically designed for this task such that it differs from FOP as summarized in Table 1.

5 Evaluation

In order to evaluate DOP in practice, we have implemented a set of case studies in DELTAJAVA that have also been studied in the context of JAK [10]. These case studies include two versions of the EPL [1,25], two smaller case examples [7], and the Graph Product Line (GraphPL), suggested in [24] as a benchmark to compare SPLs architectures. The first implementation of the EPL in AHEAD [1] follows the derivative module principle. The second implementation of the EPL is the same as sketched in this paper and presented in [25]. In the corresponding DELTAJAVA implementations, the design of the delta modules has been chosen to mimic the AHEAD design. In order to evaluate the flexibility of DOP, we have implemented each example in DELTAJAVA starting from a simple core product and from a complex core product. For the EPL, we used the simple and the complex core products presented in Section 3.

The results of our evaluation are summarized in Table 2 containing the number of feature modules or delta modules and the corresponding lines of code required to implement the respective examples. The number of feature modules and delta modules does not differ significantly in the considered examples. For the first version of the EPL [1],

Table 2. Evaluation Results (LOC is the number of lines of code)

	JAK		DELTAJAVA Simple Core		DELTAJAVA Complex Core	
	# feature modules	LOC	# delta modules	LOC	# delta modules	LOC
EPL [1]	12	98	7	123	6	144
EPL [25]	6	98	5	117	5	124
Calculator [7]	10	75	6	76	6	78
List [7]	4	48	3	58	2	59
GraphPL [24]	19	2348	20	1407	19	1373

the only reason that 12 features modules are necessary is that also interfaces are implemented by separate modules which is a design decision taken in [1]. In the second version of the EPL [25], the number of delta modules plus the core module is actually the same as the number of features modules, since DELTAJAVA encodes the same modular SPL representation. In the Calculator and List examples, less delta modules are required because several feature modules could be combined into one delta module, whereas in the GraphPL example with the simple core, additional delta modules are used to factor out common code for combinations of features. In the considered examples, the differences between the number of delta modules required to implement a SPL starting from a simple core or starting from a complex core are marginal. A more conceptual analysis on how the choice of the core product influences the SPL design is subject to future work.

In the smaller case examples, the lines of code in DELTAJAVA exceed the lines of code required in JAK, because in DELTAJAVA the feature model encoding and the application conditions have to be specified. Furthermore, as DELTAJAVA currently has no call to the original variants of modified methods, the required renaming has to be done manually, leading to additional lines of code, in particular for the EPL. This can be avoided if DELTAJAVA is extended with an operation similar to the JAK Super() call as pointed out in Section 4. In the larger case example of the GraphPL [24], delta modules require much less code, because they can represent product functionality more flexibly. First, common code for two features can be factored out into one delta module, and second, combinations of features can be treated directly by designated delta modules instead of duplicating code for feature combinations. This shows that DOP can be beneficial in terms of code size for larger SPLs in which the optional feature problem [20] arises. However, tool support has to be provided to deal with the complexity that is introduced by the flexibility of DOP, e.g., for visualizing dependencies between delta modules applicable for the same feature configuration.

6 Related Work

The approaches to implementing SPLs in the object-oriented paradigm can be classified into two main directions [19]. First, annotative approaches, such as conditional

compilation, frames [36] or COLORED FEATHERWEIGHT JAVA (CFJ) [17], mark the source code of the whole SPL with respect to product features on a syntactic level. For a particular feature configuration, marked code is removed.

Second, compositional approaches, such as DELTAJAVA, associate code fragments to product features that are assembled to implement a particular feature configuration. In [25], general program modularization techniques, such as aspects [18], framed aspects [26], mixins [33], hyperslices [34] or traits [15,11], are evaluated with respect to their ability to implement features. Furthermore, the modularity concepts of recent languages, such as SCALA [28] or NEWSPEAK [12], can be used to represent product features. Although the above approaches are suitable to express feature-based variability, they do not contain designated linguistic concepts for features. Thus, DOP is most closely related and compared to FOP which considers features on a linguistic level. Apart from JAK [10], there are various other languages using the FOP paradigm, such as FEATUREC++ [4], FEATUREFST [5], or Prehofer's feature-oriented JAVA extension [30]. In [27], CAESARJ is proposed as a combination of feature modules and aspects extending FOP with means to modularize crosscutting concerns.

The notion of program deltas is introduced in [25] to describe the modifications of object-oriented programs. In [32], DOP is used to develop product line artifacts suitable for automated product derivation and implemented with frame technology [36]. In [31], delta-oriented modeling is extended to a seamless model-based development approach for SPLs where an initial product line representation is stepwise refined until an implementation, e.g., in DELTAJAVA, can be generated. The ordering of delta modules within the `after` clause resembles the precedence order on advice used in aspect-oriented programming, e.g., in ASPECTJ [21]. The constraints that are generated for delta modules in order to ensure safe product generation require the existence and non-existence of classes, methods or fields which is similar to the constraints used in [22]. Delta modules are one possibility to implement arrows in the category-theoretical framework for program generation proposed by Batory in [8].

7 Conclusion and Future Work

We have presented DOP, a novel programming approach particularly designed to implement SPLs. It allows the flexible modular implementation of product variability starting from different core products. Because core products are complete product implementations, they can be developed with well-established single application engineering techniques to ensure their quality. DOP provides a solution to the optional feature problem [20] by handling combinations of features explicitly.

For future work, we will extend DELTAJAVA with a `Super()` call as in JAK to directly express the access to methods that are modified by delta modules applied later during product generation in order to avoid a combinatorial explosion for combinations of optional features. Furthermore, we will improve the tool support for DELTAJAVA with IDE functionalities, e.g., to show the set of applicable delta modules for a given feature configuration. In order to propose a process for the selection of core products, we are investigating how to the choice of the core product influences the design of the delta modules. Finally, we are aiming at efficient verification techniques of SPLs implemented by core and delta modules without generating the products. This work will use

the information available in the delta modules to determine unchanged parts between different products to reuse verification results.

Acknowledgements. We are grateful to Sven Apel, Don Batory and Roberto E. Lopez-Herrejon for many insightful comments on a preliminary version of this paper. We also thank the anonymous SPLC referees for detailed suggestions for improving the paper. This work has been partially supported by MIUR (PRIN 2009 DISCO) and by the German-Italian University Centre (Vigoni program 2008-2009). Ina Schaefer's work has been partially supported by the Deutsche Forschungsgemeinschaft (DFG) and by the EU project FP7-ICT-2007-3 HATS.

References

1. Expression Problem Product Line, Webversion, http://www.cs.utexas.edu/users/schwartz/ATS/EPL/
2. Apel, S., Kästner, C., Grösslinger, A., Lengauer, C.: Type safety for feature-oriented product lines. Automated Software Engineering An International Journal (2010)
3. Apel, S., Kästner, C., Lengauer, C.: Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In: GPCE, pp. 101–112. ACM, New York (2008)
4. Apel, S., Leich, T., Rosenmüller, M., Saake, G.: Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 125–140. Springer, Heidelberg (2005)
5. Apel, S., Lengauer, C.: Superimposition: A language-independent approach to software composition. In: Pautasso, C., Tanter, É. (eds.) SC 2008. LNCS, vol. 4954, pp. 20–35. Springer, Heidelberg (2008)
6. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
7. Batory, D.: A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite (ATS). In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 3–35. Springer, Heidelberg (2006)
8. Batory, D.: Using modern mathematics as an FOSD modeling language. In: GPCE, pp. 35–44. ACM, New York (2008)
9. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Trans. Softw. Eng. Methodol. 1(4), 355–398 (1992)
10. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Eng. 30(6), 355–371 (2004)
11. Bettini, L., Damiani, F., Schaefer, I.: Implementing Software Product Lines using Traits. In: SAC, OOPS Track, pp. 2096–2102. ACM, New York (2010)
12. Bracha, G.: Executable Grammars in Newspeak. ENTCS 193, 3–18 (2007)
13. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison Wesley Longman, Amsterdam (2001)
14. Delaware, B., Cook, W., Batory, D.: A Machine-Checked Model of Safe Composition. In: FOAL, pp. 31–35. ACM, New York (2009)
15. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.: Traits: A mechanism for fine-grained reuse. ACM TOPLAS 28(2), 331–388 (2006)
16. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon Software Engineering Institute (1990)

17. Kästner, C., Apel, S.: Type-Checking Software Product Lines - A Formal Approach. In: ASE, pp. 258–267. IEEE, Los Alamitos (2008)
18. Kästner, C., Apel, S., Batory, D.: A Case Study Implementing Features Using AspectJ. In: SPLC, pp. 223–232. IEEE, Los Alamitos (2007)
19. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in Software Product Lines. In: ICSE, pp. 311–320. ACM, New York (2008)
20. Kästner, C., Apel, S., ur Rahman, S.S., Rosenmüller, M., Batory, D., Saake, G.: On the Impact of the Optional Feature Problem: Analysis and Case Studies. In: SPLC. IEEE, Los Alamitos (2009)
21. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
22. Kuhlemann, M., Batory, D., Kästner, C.: Safe composition of non-monotonic features. In: GPCE, pp. 177–186. ACM, New York (2009)
23. Liu, J., Batory, D., Lengauer, C.: Feature oriented refactoring of legacy applications. In: ICSE, pp. 112–121. ACM, New York (2006)
24. Lopez-Herrejon, R., Batory, D.: A standard problem for evaluating product-line methodologies. In: Bosch, J. (ed.) GCSE 2001. LNCS, vol. 2186, pp. 10–24. Springer, Heidelberg (2001)
25. Lopez-Herrejon, R., Batory, D., Cook, W.: Evaluating Support for Features in Advanced Modularization Technologies. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 169–194. Springer, Heidelberg (2005)
26. Loughran, N., Rashid, A.: Framed aspects: Supporting variability and configurability for aop. In: Bosch, J., Krueger, C. (eds.) ICOIN 2004 and ICSR 2004. LNCS, vol. 3107, pp. 127–140. Springer, Heidelberg (2004)
27. Mezini, M., Ostermann, K.: Variability management with feature-oriented programming and aspects. In: SIGSOFT FSE, pp. 127–136. ACM, New York (2004)
28. Odersky, M.: The Scala Language Specification, version 2.4. Technical report, Programming Methods Laboratory, EPFL (2007)
29. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering - Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
30. Prehofer, C.: Feature-oriented programming: A fresh look at objects. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 419–443. Springer, Heidelberg (1997)
31. Schaefer, I.: Variability Modelling for Model-Driven Development of Software Product Lines. In: Intl. Workshop on Variability Modelling of Software-Intensive Systems (2010)
32. Schaefer, I., Worret, A., Poetzsch-Heffter, A.: A Model-Based Framework for Automated Product Derivation. In: Proc. of MAPLE (2009)
33. Smaragdakis, Y., Batory, D.: Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. ACM Trans. Softw. Eng. Methodol. 11(2), 215–255 (2002)
34. Tarr, P., Ossher, H., Harrison, W., Sutton Jr., S.M.: N degrees of separation: multi-dimensional separation of concerns. In: ICSE, pp. 107–119 (1999)
35. Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe Composition of Product Lines. In: GPCE, pp. 95–104. ACM, New York (2007)
36. Zhang, H., Jarzabek, S.: An XVCL-based Approach to Software Product Line Development. In: Software Engineering and Knowledge Engineering, pp. 267–275 (2003)

Architecting Automotive Product Lines: Industrial Practice

Håkan Gustavsson[1] and Ulrik Eklund[2]

[1] Scania, Södertälje, Sweden, Hakan
Hakan.Gustavsson@scania.se
[2] Volvo Car Corporation, Göteborg, Sweden
UEklund@volvocars.com

Abstract. This paper presents an in-depth view of how architects work with maintaining product line architectures in the automotive industry. The study has been performed at two internationally well-known companies, one car manufacture and one commercial vehicle manufacture. The results are based on 12 interviews with architects performed at the two companies. The study shows what effect differences such as a strong line organization or a strong project organization has on the architecting process. It also shows what consequence technical choices and business strategy have on the architecting process. Despite the differences the results are surprisingly similar with respect to the process of managing architectural changes as well as the information the architects maintain and update, especially in the light that the companies have had no direct cooperation.

Keywords: Architecting, Process, Case study, Automotive industry.

1 Introduction

Software and electronics are today an important part in the development of automotive products. Experts [1] estimate that 80 percent of all future automotive innovations will be driven by electronics. Scania [2] claims that electronics in trucks and buses makes up 10-15 percent of the value and is increasing. Volvo Cars [3] estimates the value of electronics of a high-end car to 30 percent.

Architectural changes of distributed embedded systems are either evolutionary or revolutionary [4], and the architecture plays a vital role to the success of the product line. The main purpose of this paper is to understand how architecting is performed to keep up with evolutionary changes. This is summarized in the research question to be answered: What tasks are performed in the process of architecting automotive embedded systems?

Decisions in the development process [5] and within the architecting process [6] has been previously studied. Dobrica and Niemela [7] makes a comparison of eight different available software architecture analysis methods. Experience reports of introducing product lines in the automotive domain for the first time has been done previously [8] as well as showing the benefits of the introduction [9]. In a survey of 279 IT architects in the Netherlands Farenhorst et al. [10] concludes that architects are

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 92–105, 2010.

lonesome decision makers; not very willing to share architectural knowledge, but eager to consume.

This paper presents a comparison of how architects at two different companies work with maintaining existing product lines. The case study has been performed at two automotive companies, the truck and bus manufacturer Scania and the car manufacture Volvo Cars. In the next section a brief presentation is given of a general automotive electrical system. In Sec. 3 the method used in the study is presented. An outline of the case study is given in Sec. 4 followed by the results in Sec. 5. Finally we discuss the findings from our work.

2 Background

2.1 The Systems and Their Architecture

The electrical system in both cars and trucks/buses are an embedded software system consisting of 30-70 different Electronic Control Units (ECUs), each with a microprocessor executing in the order of 1 MByte compiled code[1]. These ECUs control the behavior of virtually all electrical functions, from power windows to valve timing of the engine. The in-vehicle software share a number of characteristics common to the automotive domain (see e.g. [11], [12] and [13] for further elaboration):

- A large number of vehicle models with varying feature content and configurations which must be supported by the software
- Highly distributed real-time system
- Distributed development at vehicle manufacturers and suppliers
- Low product cost margins
- Stringent dependability requirements

This combination of characteristics, together with a steady growth of features realized by electronics and software, makes the electrical system in a vehicle a highly complex software system.

Almost all ECUs have a number of sensors and actuators connected to them depending on purpose and location, and these can be shared among distributed functions. Most ECUs are reprogrammable, i.e. has flash memory and not ROM, which allows programming both in the manufacturing plant as well as at dealers and workshops after delivery to the end-user. The layout of which ECUs are connected to which bus and what ECUs are acting as communication gateways between the buses is the network topology of a vehicle, of which Fig. 1 is a representative example. The interface between the software application on each ECU is in a Scania vehicle defined by the J1939 standard [14], which is very detailed in what information is. Volvo Cars uses a proprietary solution for the multiplexed communication which allows a high degree of flexibility in defining and maintaining interfaces on the buses [15]. Much of the activities regarding the logical architecture at both companies are focused on these interfaces.

[1] A few safety-critical ECUs have two microprocessors for redundancy or internal monitoring.

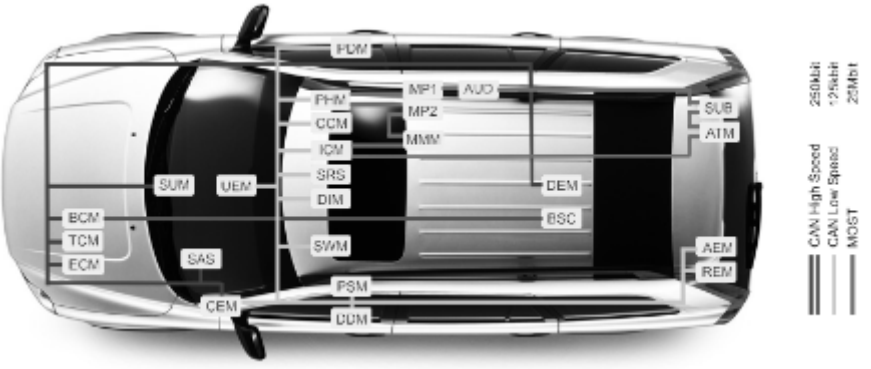


Fig. 1. The network topology of a Volvo XC90. The ECUs connected to CAN and MOST and the main multiplexed networks are seen in their approximate physical location. See [16] for a more in-depth description of the network topology of both Scania and Volvo vehicles.

2.2 Related Work

Almost all of the cases we found regarding product lines focused either on the prerequisites for a successful product line approach or the change management of an organization adapting a product line where it previous not had one. Some examples from the automotive industry are [17], [8] and [18].

Buhrdorf et al. [19] reports about the transition Salion did to a product line with a reactive approach where the necessary variations was not explored when introducing the product line, but rather handled in what they call the "steady state". The architecting work in this paper is also reactive with the same definition, since it is about updating the systems and their architectures to comply with prerequisites not known when the architecture was first designed.

3 Methodology

The data used in this study is based on interviews with the persons most involved in the activities of maintaining architectures, i.e. the architects themselves. Neither Scania nor Volvo makes a distinction of the roles for system and software architects. All architects available and willing to participate were interviewed, which resulted in more than half of the architects at each company participating, 4 at Scania and 5 at Volvo Cars. In addition to this the managers for the architecture groups were interviewed at both companies, totaling the number of interviews to 11. Of the 11 respondents 2 were women.

The interviews were performed by the two authors, which are native to Scania and Volvo Cars respectively (see [20] for the definition of "native" in this context). One lead the interview while the other took extensive notes, which was later edited for spelling and grammar. The respondents had the possibility to read and comment the notes from their respective interview to correct any misunderstandings, purse errors or other mistakes in the recordings. This was done before the analysis took place.

The interviews were semi-structured with open-ended questions. The researchers paid special attention to not use any terminology that had special or different meanings at the two companies to avoid the respondents perceive the same question differently depending in which company they were working. After the interview was constructed, it was tested on one person at each company who had worked as a system architect to evaluate the relevance.

The interview questions were defined in English and then translated to the native language of the interviewers and respondents, Swedish, for a more natural and fluent setting. Whenever a quote from the interviews is presented in the article the translation to English was done post mortem.

The interview started with some introductory questions to get some background about the respondent, like education, professional experience of embedded systems, time employed and a general idea of how they would define architecture. The majority of each interview was based on a set of questions directed at exploring the respondent's view of their work with the architecture. The set of questions were aimed to cover all stages of an architecting process from [21] to make sure no vital information was missed. All 11 interviews progressed in essentially the same order.

3.1 Analysis Procedure

The analysis was made by the two researchers jointly looking for common themes based on the interview questions. Also answers relating to these themes given in other questions were including in this analysis. The themes were also analyzed if they showed a close similarity between the two companies or significant differences. The two authors used their insider knowledge about respective organization and products in making the analysis and to enrich the conclusions made.

4 The Case Study

The main objective of this study was to get the richest insight possible into how architects maintain an existing architecture in practice. The selection of the two automotive companies was made for three reasons. The first is that the authors already had inside access to the subjects and the support of middle management to perform this and similar studies. Second the two companies are similar enough for a comparison to be manageable, such as each company having a product line architecture approach, but still different enough for the interviews not to be a duplicate. The third, and not least, reason is the possibility for the authors to use their knowledge as insiders to augment the analysis of the data to provide an even richer insight into the two cases.

4.1 Context

Both companies studied are situated in Sweden and share characteristics common among Swedish engineering industries such as; solid knowledge about the product among the developers, putting value on personal networks, and similar educational and demographic background in the development departments. The overall product development process at both companies follows a traditional stage-gate

model. An important difference is the balance of power; Scania has a stronger line organization [22] while at Volvo Cars the project organization is stronger.

All participants had a similar educational background with an engineering master degree from a Swedish university. They had worked with embedded systems between 5 and 25 years. They also had similar experience working as architects, with a majority being an architect for 4-6 years. The main difference was that the architects at Volvo Cars had on average worked twice as long in the company, compared to Scania.

Scania is one of the world's leading manufactures of heavy commercial vehicles selling on a global market with a solid reputation of designing and producing vehicles with the core values of "Customer first", "Respect for the individual" and "Quality". During 2008[2] Scania produced 66,516 trucks and 7,277 buses. Scania is a public company with Volkswagen AG as the largest stockholder. The development of all critical parts of the product, such as engine, transmission, cabs and chassis are centralized to the research and development centre in Södertälje, Sweden.

Volvo Car Corporation is a manufacturer of premium cars with core values[3] of "safety", "environment" and "quality". Volvo Cars produced 374,297 vehicles in 2008[4]. Volvo Cars is a subsidiary company to Ford Motor Company (as of 2010 February 23), sharing technical solutions with other brands within FMC.

4.2 The Scania Product Line

Scania has a tradition of working with a modular product design since the early 1960's. The modular system has claimed to be the main reason why the company stayed profitable every year since 1934 [23]. The internal training program teaches the three basic corporate principles of modular thinking [24]:

1. Standardized interfaces between components
2. Well-adjusted interval steps between performance classes
3. Same customer-need pattern = same solution

These principles are today also applied on the electrical and electronic system, besides the traditional mechanical parts. Scania does all design work towards the product line, there is no work done towards a specific product model. A project at Scania is an addition or update to one or more modules towards a specific time when it goes into production, and there is no difference if the update is purely mechanical or includes software as well, the product line approach is identical [24]. The Scania product line uses the same architecture, as well as components, for all of its three product categories; trucks, buses and engines, seen in Fig. 2. Every sold product is customer ordered and unique which is made possible through the modular system.

---

[2] http://www.scania.com/scania-group/scania-in-brief/key-figures/

[3] http://www.volvocars.com/intl/top/about/values/pages/default.aspx

[4] http://www.volvocars.com/SiteCollectionDocuments/TopNavigation/About/Corporate/VolvoSustainability/VolvoCars_report_2008_ENG.pdf

The software adaptation of each product is made during production. This is done by extracting a configuration file from the manufacturing product specification, which is then downloaded onto the unique product.

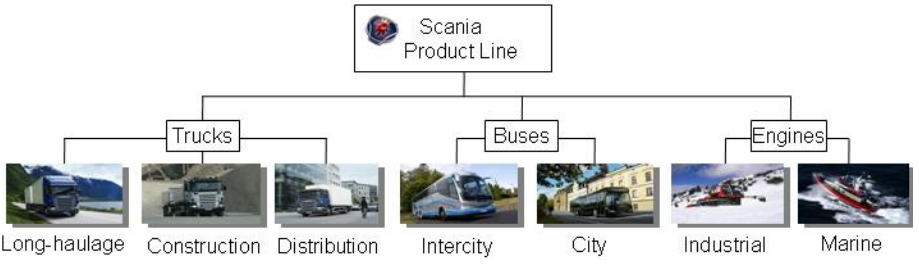


Fig. 2. The product line at Scania and the different products built on it

4.3 The Volvo Cars Product Lines

Presently Volvo Cars maintains 3 electrical architectures for the 3 platforms in production. All vehicles in a platform are said to share the same architecture, which includes the software as well as the hardware it is executing on.

Volvo does most engineering work towards a new vehicle model, or model year, but with the intention that a solution should later be used for other vehicles on the same platform. In contrast to Scania, Volvo defines the product requirements for the individual car models and not the product line as a whole. The development of the architecture and sub-system solutions are shared between the platform and the individual products, an approach driven by the developers at the Electrical and Electronic Systems Engineering department themselves rather than a company-wide business strategy.

All vehicles produced are made to order. With the possibility for the customer to select optional features and packages the theoretical number of possible software configurations surpasses the actual number built by orders of magnitudes.

4.4 Comparison of the Product Lines

Both companies can be said to have a product line, including both hardware and software, and how they develop and maintain architectures. The electrical system share a common set of features aimed at a particular market segment, e.g. premium cars or heavy commercial vehicles, and is developed from a common set of assets (e.g. a common architecture and shared systems between vehicle models). The architectures prescribe how these shared systems interact. Since these criteria are fulfilled the software are a software product line according to [9].

The two approaches to product lines were not driven by a business decision but by the development organizations adapting to their environment. Both companies were also early adopters of the practice of building several different vehicles on the same manufacturing line, implemented years before the introduction of complex electrical systems.

Supporting factors for establishing a product line of the electrical system were in Volvo's case having a rather narrow spread in vehicle models together with an explicit single options marketing strategy (versus fixed packages). This lead to a system with a high degree of configurability. In Scania's case the supporting factors were the organization wanting to develop vehicles tailored to their customers, maximizing customer value without having to redo similar development work over and over again.

Both companies handle variability in very similar way. The architecture is predominantly implemented with two mechanisms according the taxonomy by Svahnberg et al [25]: Binary replacement—physical, where different binaries can be downloaded to the flash memory of all ECUs depending on the configuration of customer-chosen optional features such as adaptive cruise control. This can be done in the manufacturing plant using the plant's product data system with separate article numbers for software as well as hardware (including nuts and bolts) and in the aftermarket using proprietary systems. At Volvo this is accomplished by the Product Information Exchange system for software [26].

The most common variability mechanism is *Condition on variable* where all ECUs get information from a central on-board file defining the configuration of that vehicle. This file is generated automatically in the manufacturing plant and flashed as a separate binary to a central ECU. Some ECUs also store local variables similarly used in a separate binary file with its own article number as well.

5 Results

The interviews yielded results mostly regarding the process for managing an architectural change.

5.1 The Process

The process for managing changes to the architecture is very similar at the two organizations with five distinct activities:

1. need
2. impact analysis
3. solution
4. decision
5. validation

This is a fairly general process, easily mapped to a generic process for architecture work seen in Fig. 3, based on [21].

At Volvo Cars there is a greater emphasis on “why” the architecture needs to be changed, as described by one of the architects on what is done first:

> “Do a need analysis on what is driving the change. What isn't good enough? What change is needed?”

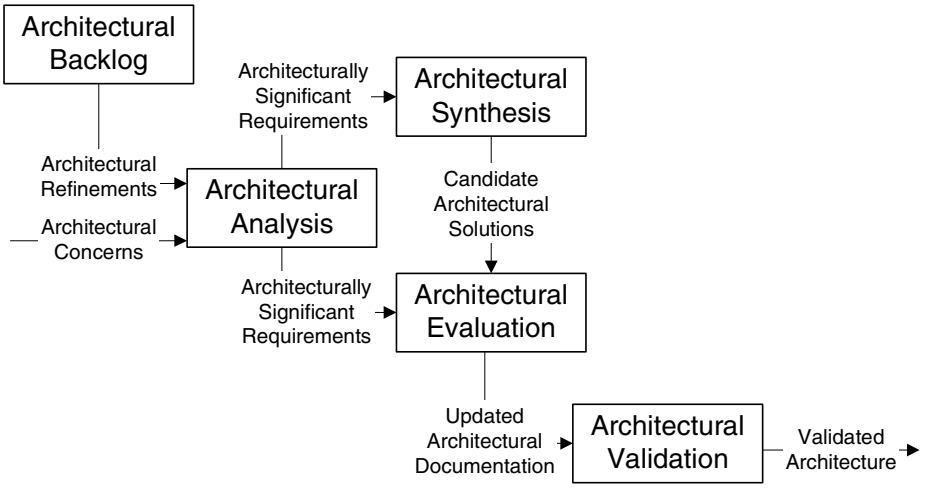


Fig. 3. A generic process for creating and maintaining an architecture, adapted from [21]

At Scania the architects' focus is on "how", i.e. the impact of an architectural change. One possible conclusion is that the "why" is seen as a strategic responsibility of the senior architect at Scania, and the other architects are more concerned with "how". Another possible reason is that the Scania architecting group has chosen to be more supportive than controlling.

5.2 Needs to Change the Architecture

Architects at both companies mention functional changes and functional growth as common reasons to update the architecture. This is not surprising since most new features are realized by electronics and software, and that the number of features grows almost exponentially [13].

At Volvo Cars all architects mention cost or cost reduction as a common reason to change the architecture, this is not surprising since the cost margins are very small and if an opportunity presents itself it is considered. At Scania cost was only mentioned by the manager, and then only in the context of how much the architectural change would cost. The most common reason to change the architecture mentioned by the architects at Scania was to adapt it to hardware changes, as described by one Scania architect: "Control units become too old; there is no room for development".

5.3 Architecture Impact Analysis

The architects at Scania clearly seek to identify who is concerned by a change and what parts of the system are impacted by a proposed change. At Volvo Cars the architects request information about non-functional requirements or quality attributes and use cases when analyzing the impact, as described by one architect:

> "I need a good description of what the customer should expect form the system. If it concerns a ready solution or if it is something we should develop internally, it

> could be a supplier offering something which we should integrate in the system. If there is a system solution which should be integrated I want to see that as well, if there are variants and if it is to be sold as option or standard. . . "

A possible explanation to this could be that the architects at Scania are involved earlier in the development of new features or systems, while at Volvo Cars the architects are more often given a proposed technical solution, for example by a supplier. The managers at both Scania and Volvo Cars mentioned the motive for the change as important information for understanding the change, but no other architects mentioned this. We have no explanation why this is so...

The time it takes to understand the impact on the architecture from a change seems to be similar between the two companies, a few weeks to a month calendar time. It seems to depend more on finding the right stakeholders and set up appointments with them than the actual effort in man hours from the architects. Some architects at Volvo Cars also say some architectural changes takes only minutes to evaluate the impact. This could be explained by fact that such a question would not require a official Change Request at Scania and therefore the respondents have not included these issues in their answers, or that the architects at Volvo Cars usually have a more final solution to evaluate.

5.4 Design Alternatives

Not very surprising, but notable no architecture analysis methods [7] were used or mentioned. Evaluation was in rare cases made using methods very similar to Pugh evaluation matrix [27]. Volvo architects seem to more evaluate how well different design alternatives fit into the present architecture, as mentioned by one of the architects:

> "Put some different alternatives against each other and evaluate from different aspects which is best. Cost is one example. Often the need does not come from the architecture, but from different sub-systems, from the outside. When you know what needs to be done the implementation phase begins. I follow long into the project and follow up that verification is done."

In comparison to this the Scania architects are more involved in developing different alternatives in the modelling activity. The architects see themselves as having a supporting role to function and sub-system developers. This is exemplified by

> "Requirements on new functionality are often what we start with. We then balance that against the present architecture, layout of electronics and the electrical system and weigh it against our (architectural) principles. How can we enable the functionality? Sometimes it is easy to fit in and sometimes we realize we don't have the necessary hardware and that requires a bigger effort and we go through a number of steps."

This difference in how involved the architects are in the development of subsystems is probably driven by Volvo Cars having a much larger percentage of purchased sub-systems than Scania.

5.5 Deciding on the Architectures

Architects at both companies stated that most (all?) decisions when updating the architecture were driven by non-functional requirements, quality attributes or constraints. However the attributes differed between the two companies even though the products are fairly similar, trucks/buses versus cars. The attributes deciding what update to make to the architecture could in most cases be derived from the core values for each company, for Scania Customer First, Respect for the Individual and Quality, and for Volvo Cars Safety, Environment and Quality. The attributes mentioned by Scania architects were time (to implementation), personnel resources, system utilization, including network bus load, safety, evolvability, usability, robustness, maintainability and commercial effectiveness (of which cost is a factor).

The architects at Volvo Cars unanimously mention cost as the most important factor when deciding between architectural alternatives. Other factors they mention are if the solution can realize the desired functionality, time and resources for implementation, environment friendliness exemplified by current consumption, weight, network bus load, including timing aspects, driveability, comfort and safety requirements. Risk, or minimizing the risk of a change, was also mentioned as a constraint by Volvo architects. The risk of change was not mentioned at Scania, possibly due to being obvious to think about.

A common constraint, which was mentioned by architects at both companies, was a clear wish of minimizing the effect of any architectural changes to any already existing sub-systems. The architects usually made a point of considering how a change would affect all sub-systems and not only the one proposing the change. There was a common architectural concern to have as small changes as possible, to quote one architect from Volvo Cars:

> ". . . if we need to compromise so much it hurts we have not done a good job. If we don't need to compromise so much it is good."

5.6 Validation

The most interesting result found was that none of the architects at the two companies validated the result of the implemented change themselves. Many of the architects at Scania had a clear idea of which stakeholder they would get feedback from, the integration test group. The architects at Volvo Cars were more vague when expressing how they follow up an architectural change:

> "If it isn't a good solution we get to know there is a problem which we correct. Normally we assume that testing finds (anything)."

Common between the two companies was that the architects mentioned review of specifications on how a change in the architecture is followed up, but it is unclear exactly what documents the architects are reviewing.

5.7 The Resulting Artefacts from the Architects' Work

The resulting artefacts from the architects' work on the changes to the architecture are very similar between Scania and Volvo Cars. It is the responsibility of the architects

to update the network topology if a requested change affects how and where an ECU is connected to a network. At Volvo Cars the view of the topology is part of the officially released Architecture Description, one for each platform or product line, which is edited by the architect for the platform. At Scania the view of the topology is a separate document which is updated at every new release.

At both companies there will be a model describing the logical architecture captured in an UML tool. At Scania this model grows when a change concerns an area or function not previously modelled. Volvo Cars already has a more comprehensive model covering the complete existing system, so if the feature is not completely new it is more of a question of updating the existing model. Another artefact that gets updated is the signal database mentioned above. At Scania the architects defines message sequence charts (MSC) defining the interaction between ECUs, something that is not done at all by the architects at Volvo Cars.

The general conclusion is that the architects at both companies work with essentially the same type of information, but packaged slightly differently. Meetings are more emphasized at Scania, as stated from one of the architects;

> "…there is more eye-to-eye communication than document communication compared to other companies I have worked at."

5.8 The Timing

The timing of when a change is introduced in the architecture varies and is driven by different factors at the two companies. At Scania the most important factor mentioned is when all concerned developer stakeholders are able to update their design. All concerned developers synchronize the changes of their assets in the product line towards a common start-of-production (SOP). These change projects are tracked on visual planning boards [28].

The timing of architectural changes at Volvo Cars is usually driven by the project timing for launching new car models (also called start-of-production at Volvo Cars), or updating a new year model of an existing car. The architects respond to these change requests if they are technically possible to do within that time frame. However, in the interviews two architects expressed hesitation when claiming that it was only the project that determined the timing. To summarize: At Scania the timing of a change of the architecture is determined by the contingence of the line organization while at Volvo Cars it is determined by the need of the vehicle model project.

5.9 Other Observations

The architects at Volvo Cars had on average worked twice as long in the company, while all architects at Scania except one had worked 4 years or less at the company. The conclusion is that at Volvo Cars the architects were recruited internally form other roles while at Scania the architects were employed specifically into that role. One noticeable difference to this is the senior architect at Scania with 21 years in the company; he is also the only one of the 11 interviewees with an official recognition as senior or expert in the two organizations.

The difference in work tasks between Scania and Volvo Cars is that at Scania the architects usually works with a specific domain, e.g. HMI or chassis systems, while at Volvo the architects were responsible for a platform and the entire system on it, e.g. the large platform (S80, V70, XC60, . . .).

6 Discussion

The striking conclusion and the answer to the stated research question is the similarity between the two companies in the tasks performed when maintaining and changing architecture. The *tasks* mentioned by the architects at both companies are virtually identical; need ⇨ impact analysis ⇨ solution ⇨ decision ⇨ validation.

The tasks do not seem to be different for architecture maintenance compared to developing a new architecture. Likewise they seem to be the same whether it is updating a product line architecture or updating the architecture of a single-shot system. Also the types of information the architects work with, one could say the viewpoints, is almost identical between the two companies. The difference being sequence charts are only used at one company but there the architects say they maintain them as a service to other stakeholders and they are not architecturally relevant. The description of the architects as lonesome decision makers made by Farenhorst et al. [10] could not be seen in this study. One possible reason for this could be the cultural differences between Sweden and the Netherlands.

The similarity in process and information is surprising since the present processes of the two companies have evolved almost independently at respective company. The similarities could be explained by the systems in cars and commercial vehicles are similar and that the companies are not too different in the demographics of their architects in terms of experience, education etc. One reason could be that the processes found can easily be mapped to a general process for architecture work, as found in [21].

As shown by Nedstam [6] there is large difference of how work is done in an organization with strong line management and a organization with strong projects. Several of the observed differences between the two companies could have affected how they work with architectural change, such as the differences in their product line approaches, the focus on project versus line organization and differences in quality attributes.

The fact that Volvo Cars has a higher degree of tool support while Scania are more conscious with respect to processes was also expected to affect the work of the architects more than was found in this study.

Acknowledgments. We would like to thank the architects and managers at the two companies for their interest and cooperation.

This work has been financially supported by the Knowledge Foundation and the Swedish Agency for Innovation Systems (VINNOVA) as part of the FFI program.

References

1. Grimm, K.: Software Technology in an Automotive Company - Major Challenges. In: International Conference on Software Engineering, pp. 498–503 (2003)
2. Edström, A.: Hasse vill ha mer processorkraft. Elektroniktidningen, 26–29 (2008)
3. Edström, A.: Urban på Volvo hyllar säkerheten. Elektroniktidningen (2006)
4. Axelsson, J.: Evolutionary Architecting of Embedded Automotive Product Lines: An Industrial Case Study. In: Rick Kazman, F.O., Poort, E., Stafford, J. (eds.) Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) & European Conference on Software Architecture (ECSA 2009), pp. 101–110 (2009)
5. Gustavsson, H., Sterner, J.: An Industrial Case Study of Design Methodology and Decision Making for Automotive Electronics. In: Proceedings of the ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, New York (2008)
6. Nedstam, J.: Strategies for management of architectural change and evolution. Lund University, Department of Communication Systems, Faculty of Engineering, Lund (2005)
7. Dobrica, L., Niemela, E.: A Survey on Software Architecture Analysis Methods. IEEE Transactions on software engineering 28, 638–653 (2002)
8. Steger, M., Tischer, C., Boss, B., Müller, A., Pertler, O., Stolz, W., Ferber, S.: Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. Software Product Lines, 34–50 (2004)
9. Clements, P., Northrop, L.: Software product lines: practices and patterns. Addison-Wesley, Boston (2001)
10. Farenhorst, R., Hoorn, J., Lago, P., Vliet, H.v.: The lonesome architect. In: Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) & European Conference on Software Architecture (ECSA), pp. 61–70. IEEE, Los Alamitos (2009)
11. Schulte-Coerne, V., Thums, A., Quante, J.: Challenges in Reengineering Automotive Software, pp. 315–316. IEEE Computer Society, Kaiserslautern (2009)
12. Pretschner, A., Broy, M., Kruger, I.H., Stauner, T.: Software Engineering for Automotive Systems: A Roadmap. In: International Conference on Software Engineering, pp. 55–71 (2007)
13. Broy, M.: Challenges in automotive software engineering. In: Proceedings of the 28th international conference on Software engineering, pp. 55–71. ACM, Shanghai (2006)
14. SAE: Standard J1939 - Recommended Practice for a Serial Control and Communications Vehicle Network. Society of Automotive Engineers (2009)
15. Casparsson, L., Rajnak, A., Tindell, K., Malmberg, P.: Volcano-a revolution in on-board communications. Volvo Technology Report 1, 9–19 (1998)
16. IEEE-1471: IEEE Recommended practice for architectural description of software-intensive systems. IEEE Std. 1471-2000 (2000)
17. Voget, S., Becker, M.: Establishing a software product line in an immature domain. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 121–168. Springer, Heidelberg (2002)
18. Tischer, C., Muller, A., Ketterer, M., Geyer, L.: Why does it take that long? Establishing Product Lines in the Automotive Domain. In: 11th International Software Product Line Conference, Kyoto, Japan, pp. 269–274 (2007)
19. Buhrdorf, R., Churchett, D., Krueger, C.: Salion's Experience with a Reactive Software Product Line Approach. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 317–322. Springer, Heidelberg (2004)

20. Brannick, T., Coghlan, D.: In Defense of Being Native: The Case for Insider Academic Research. Organizational Research Methods 10, 59 (2007)
21. Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, H., Ran, A., America, P.: Generalizing a Model of Software Architecture Design from Five Industrial Approaches. In: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, pp. 77–88. IEEE Computer Society, Los Alamitos (2005)
22. Bergsjö, D., Almefelt, L.: Supporting requirements management in embedded systems development in a lean influenced. In: Proceedings of International Conference on Engineering Design, Dubrovnik, Croatia (2010)
23. Johnson, H.T., Senge, P.M., Bröms, A.: Profit beyond measure: extraordinary results through attention to work and people. Nicholas Brealey, London (2000)
24. Kratochvíl, M., Carson, C.: Growing modular: mass customization of complex products, services and software. Springer, Berlin (2005)
25. Svahnberg, M., Van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Software: Practice and Experience 35, 705–754 (2005)
26. Melin, K.: Volvo S80: Electrical system of the future. Volvo Technology Report 1, 3–7 (1998)
27. Pugh, S.: Total design: integrated methods for successful product engineering. Addison-Wesley, Wokingham (1990)
28. Morgan, J.M., Liker, J.K.: The Toyota product development system: integrating people, process, and technology. Productivity Press, New York (2006)

Developing a Software Product Line for Train Control: A Case Study of CVL

Andreas Svendsen[2,3], Xiaorui Zhang[2,3], Roy Lind-Tviberg[1], Franck Fleurey[2], Øystein Haugen[2], Birger Møller-Pedersen[3], and Gøran K. Olsen[2]

[1] ABB, Bergerveien 12, 1375 Billingstad, Norway
roy.lind-tviberg@no.abb.com
[2] SINTEF, Forskningsveien 1, Oslo, Norway
{Andreas.Svendsen,Xiaorui.Zhang,Franck.Fleurey,
Oystein.Haugen, Goran.K.Olsen}@sintef.no
[3] Department of Informatics, University of Oslo, Oslo, Norway
birger@ifi.uio.no

Abstract. This paper presents a case study of creating a software product line for the train signaling domain. The Train Control Language (TCL) is a DSL which automates the production of source code for computers controlling train stations. By applying the Common Variability Language (CVL), which is a separate and generic language to define variability on base models, we form a software product line of stations. We discuss the process and experience of using CVL to automate the production of three real train stations. A brief discussion about the verification needed for the generated products is also included.

1 Introduction

The Train Control Language (TCL) is a domain-specific language (DSL) for describing train stations in the train signaling domain. A DSL is a programming or modeling language dedicated to a particular problem domain. TCL is developed by SINTEF in cooperation with ABB, Norway, and contains a minimal but sufficient set of concepts within the train signaling domain. The purpose of TCL is to automate the production of source code that controls the signaling system on a station.

Production of TCL stations can be further automated by using software product line (SPL) technology. An SPL captures the variabilities and commonalities of a series of products that are sufficiently similar. Product line modeling involves information about all product line members, which is different from modeling a singular product.

The Common Variability Language (CVL) provides a generic and separate approach for modeling variability in models defined by DSLs such as TCL [5][4]. CVL can be applied to models in any DSL that is defined by a metamodel by means of Meta Object Facility (MOF) [7]. This paper presents a case study on how we applied CVL to TCL for developing a station product line where all the product line members are Norwegian train stations in use or under development. We report on the process of using CVL to express the variabilities and commonalities among designated products

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 106–120, 2010.

of the station product line, and how we derived and decided on the final product line based on that. We report on several issues that occurred during the development, discuss the pros and cons of different alternative solutions and report on our own experience trying out those solutions. Based on the experience of this case study, we also make some initial thoughts on the methodological support for the CVL approach and identify some open issues for future work.

The paper is structured as follows: Section 2 briefly presents the train domain, TCL and software product lines. Section 3 introduces CVL and its tool support, before Section 4 walks through the process of creating the station product line and the collected experiences from this assignment. Finally, Section 5 concludes with some open issues for future work.

2 Background

This section briefly introduces the train signaling domain and software product lines. TCL, as a DSL for this domain, was developed for the purpose of generating interlocking source code (functional blocks) for the Programmable Logic Circuit (PLC) at a station. TCL is used as the base language for our case study.

The interlocking system in the train signaling domain controls the basic elements of the station (e.g. signals, switches, track circuits etc.) and also allocate train routes in order to avoid collisions.

Fig. 1 from [9] illustrates the layout of a train station. A train route is a route between two main signals in the same direction, and it consists of several track circuits. A track circuit is the shortest distance where the presence of a train can be detected. It consists of line segments and switches connected by endpoints.

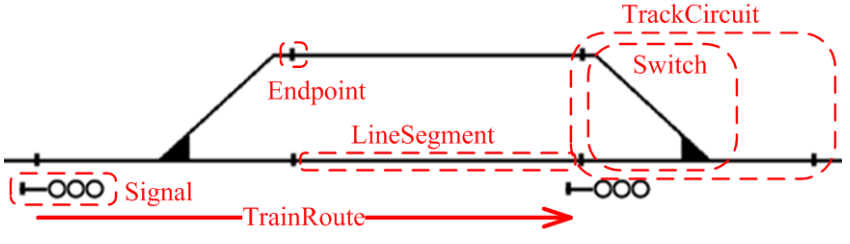


Fig. 1. Train Station Layout

2.1 The Current Process of Designing Interlocking Source Code

Svendsen et al. [10] and Endresen et al. [2] show that the current development of source code for ABB's Computer Based Interlocking (CBI) for a single station is a manual and time-consuming process. First ABB receives a structural drawing of a station with its interlocking table from the Norwegian Train Authorities, and then the train experts develop the functional specification and design specification. These are formally reviewed before two independent teams create the interlocking source code for the station based upon these specifications. This source code is then thoroughly tested.

The current development process is manual and time-consuming. This is the reason why model-driven development (MDD) is considered and TCL is developed.

2.2 Train Control Language

The TCL language is defined by an Ecore metamodel in the Eclipse Modeling Framework (EMF) [1] as explained by Svendsen et al. [10] and Endresen et al. [2]. Its tool support includes a graphical editor, a tree-view editor and code generators.

The use of TCL allows the train experts to only work on defining the station in the TCL graphical editor. The code generators will then produce other representations automatically, such as interlocking tables (truth tables) and interlocking source code. This source code, which is a form of functional blocks, is then loaded into the Programmable Logic Circuits (PLCs) for that station. The model-to-text code generator, written in MOFScript [8], requires adequate expertise in the train signaling domain and an overall understanding of interlocking source code on various train stations [10].

A station modeled by the TCL graphical editor is illustrated in Fig. 2. The figure shows the depiction of physical elements on the bottom and the more abstract concepts train route (rounded rectangles) and track circuit (rectangles) on the top. This station results in more than 3000 lines of boolean equations when generating the interlocking source code. We use this station as the starting point for our product line, which we describe in Section 4.

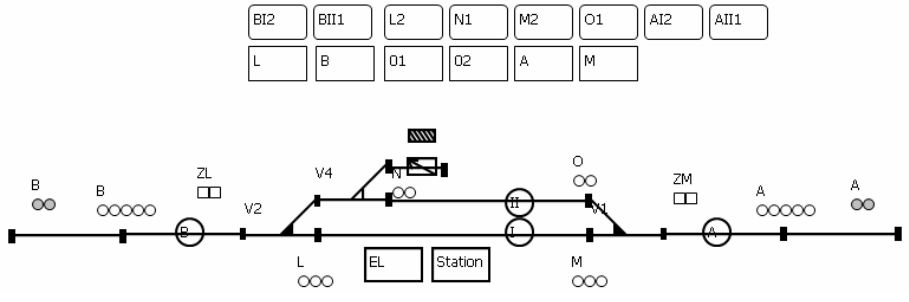


Fig. 2. Station created by the TCL graphical editor

2.3 Software Product Lines

Feature modeling as a technique for defining features and their dependencies has been widely adopted in the Software Product Line community. It was originally introduced by Kang as part of Feature-Oriented Domain Analysis (FODA) [6]. There a feature is defined for the first time as a "prominent or distinctive user-visible aspect, quality, or characteristic of a software system". Feature modeling is a means to reflect user choices and requirements in an early phase of the product design.

Features are typically modeled in the form of tree-like feature diagrams along with cross-tree constraints. A Feature Diagram is considered to be an intuitive way to visually present the user choices of the features, and is therefore widely used and extended. The FODA notation includes child features as optional, mandatory or alternative (XOR).

3 Common Variability Language

CVL [4] is itself a DSL for modeling variability in any model of any DSL based on MOF. In the CVL approach we have three models: The base model is the model described by a DSL (e.g. train stations modeled by TCL), the variability model that defines variability on the base model, and the resolution model that defines how to resolve the variability model to create a new model in the base DSL. These three models are illustrated in Fig. 3. The CVL model consists of the variability model and the resolution model. The base model is oblivious of the CVL model (there are only references from the CVL model to the base model). Several resolution models can resolve the variability in one variability model, and several variability models can describe the variability in one base model.

3.1 CVL Language

The concepts included in CVL make it possible to convey the variability into two conceptually distinctive layers: Feature specification and product realization. The feature specification layer is the user-centric part of the CVL model, and leaves the details of the connection to the base model to the product realization layer.

In the context of software product lines, the feature specification layer expresses high level features that the user would like to include, similar to feature diagrams. The concepts of CVL (e.g., type, composite variability, constraint and iterator) are sufficient to mimic feature diagrams (e.g. mandatory/optional feature, feature dependencies, XOR/OR and cardinality).

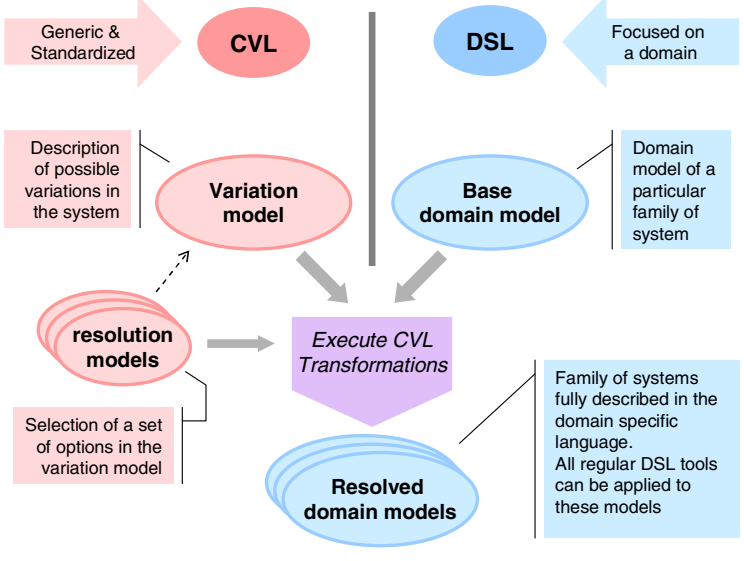


Fig. 3. CVL combining with a DSL

Based on the feature specification layer, the product realization layer further defines low level, fairly detailed, but necessary operations required to transform the base model to a resolved product model. This includes information about how base model elements are substituted for other base model elements.

CVL has concepts supporting typing of a set of model elements in the base model. With these abstraction mechanisms, the user is able to customize a set of model elements in the base model and use it to replace any compatible base model fragments.

With the two-layered conceptual distinction of the variability, CVL separates the modeling concerns as well as provides the possibility for users of different levels to understand or define a CVL model incrementally. We can think of such a scenario: the variability of the feature specification layer can be defined by domain experts of higher level design, such as to identify features of product line members, while the variability of the product realization can be defined by domain experts who are more familiar with detailed design of the system.

3.2 CVL Tool Support

The CVL tool support includes a graphical editor and a tree editor for creating and viewing the CVL model, and a model-to-model transformation to generate new resolved product models. CVL also provides a set of APIs, which can be implemented by a base language editor, to support integration between the base language editor and the CVL editor. This integration includes the possibility to create fragments (base model elements involved in a substitution) from a selection of base model elements and to highlight how base model elements are affected by a substitution. This is realized by retrieving and storing the EObject references to the base model elements.

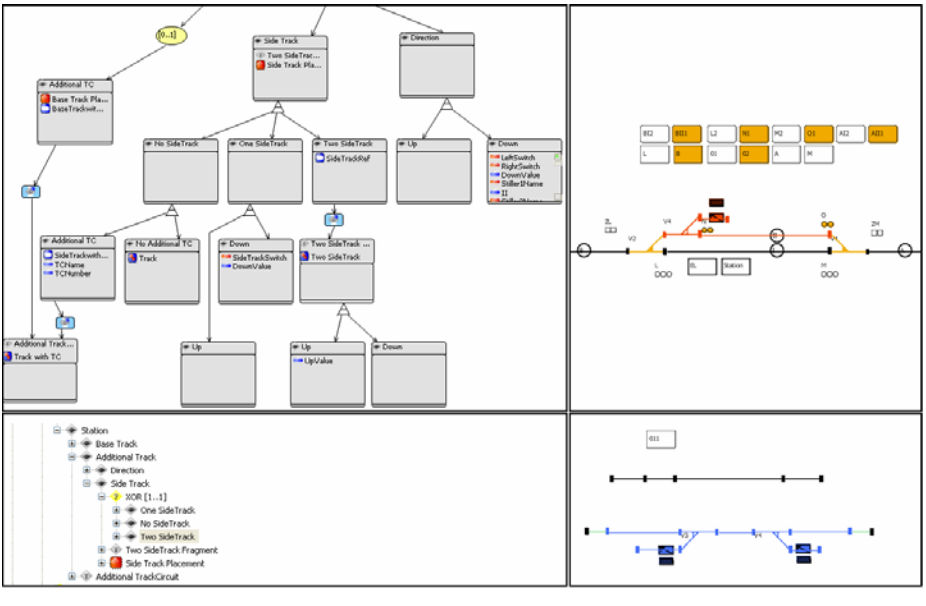


Fig. 4. CVL and TCL editor integration

As illustrated in Fig. 4, the CVL graphical editor at the top left, with the feature-diagram-like CVL diagram, and the CVL tree editor at the bottom left present the variability of the feature specification layer. The model elements involved in the variability of the product realization layer are highlighted in the base model at the top right and the base model fragment library to the bottom right. In Fig. 4 the highlighting describes a substitution where the second track with one side track (top right) is replaced by a new track with two side tracks (bottom right). Different levels of information on a CVL model can be hidden and shown when presented to people with different levels of system knowledge.

The CVL language and tool support are further explained in [4].

4 Station Product Line

TCL has automated the production of interlocking source code. For ABB, this automation does not only result in shorter time-to-market for a single station, but also in preservation of consistency and completeness by eliminating some of the error-prone manual process. However, whenever there is a need for a new station product, each station has to be modeled separately using TCL. Even though the interlocking source code is generated based on the station, the station model itself is manually created.

ABB has the need for designing several similar stations that vary slightly. Development of more than 20 stations with two or three tracks and varying number of side tracks and topology are currently being planned. Based on this need we now describe an approach for using CVL to create a product line of stations using three of these stations. The purpose of this product line is to automate the production of station models. In this section we walk through the process of creating this product line, and the collected experience from this assignment.

4.1 Preparing the Product Line

There are basically two strategies to follow when creating a product line model using CVL. The comparing strategy involves comparing specifications or models of the products to find the commonalities and differences. By using one of these models as the base model, a CVL model which describes the differences can be created. The constructive strategy involves selecting a base model as a starting point, and defining the product specifics directly in CVL. The last strategy can be advantageous if well-defined products do not exist when preparing the product line.

In our case study CVL has been applied to the train station domain, which is based on ABB's need to define more than three stations with corresponding interlocking source code. Note that these stations are real stations that have to be validated and verified. The train station domain is a static domain where the station products are well defined from the authorities. As a result, the comparison strategy is the natural choice for assembling the CVL product line model. From the structural drawings of the station products (see Section 2 for more information about the input requirement specification) the commonalities and differences were extracted. The comparison is illustrated in Table 1.

All the stations are simple two track stations, but they differ on how many side tracks (to park trains) they have, how the elements in the station are named, the number of track circuits and the direction.

Table 1. Differences and commonalities between the stations

	SideTracks	TrackCircuits on each track	Direction	Other
StationB	2	1	Down	Main track as track 2
StationD	1	1	Up	
StationS	0	2	Up	

Comparing the station drawings was pretty straightforward, but we noticed that categorizing the commonalities and differences into meaningful entities required knowledge about the domain. For instance the entity "SideTracks" in Table 1 is a collection of several elements in the station model, and it corresponds to a side track where trains can be parked. This illustrates that domain knowledge is not only necessary when developing a single station, but also in the planning and extraction of information to a station product line. Note that "SideTrack" is not a concept of TCL as such, but rather a collection of certain TCL elements. CVL can make this as an abstract concept on top of TCL (see Section 4.3).

The constructive strategy of creating the CVL model would mean to raise the abstraction level and consider what kind of functionality the product should have. In this case it would mean identifying all structures the station product line should support and using CVL to build the products properly. This could result in a more carefully constructed product line model, since all the essential structures can be included. We see that a combination of these two strategies, by starting with a comparison of a few products, but generalizing the product line to support more products, may be beneficial.

While choosing the product line members, we found it ideal to not only consider the existing station descriptions, but also include other possible products. This was, however, a trade-off of the complexity of the CVL model (see Section 4.4).

4.2 Choosing a Base Model

As presented in Section 3, the CVL model defines how a base model can vary. The execution of a CVL model will perform a model-to-model transformation from this base model to a new base model defined by the same metamodel. To define a station product line in CVL, it is required to have a station base model as a starting point.

There are several strategies for choosing such a base model. Since CVL replaces values and sets of model elements, executing CVL can add, remove or replace functionality. One obvious choice of base model can be a model with maximum set of features included, meaning a complete model where CVL can remove features to get a specific product model. In other words when we operate on a maximum base model

a subtractive strategy is used. Another choice of base model can be a model with minimum set of features included in the model itself, and other fragments in other library models (more about CVL library in Section 4.3). Then the product models will be generated by adding features to the base model. An additive strategy will thus be used when operating on a minimum base model. A third strategy is to choose a base model that has neither maximum nor minimum, but somewhere in between. This base model can for instance be the base model that is most similar to the majority of the product models, or a base model that is tailored for teaching purposes. This can be viewed as operating on an intermediate base model, where both additive and subtractive strategies will be used. One choice of an intermediate base model may be the one that results in the most compact CVL model. A compact CVL model can be measured in the number of substitutions and the complexity of the fragments used in the substitutions (e.g. number of boundary elements involved, where a boundary element in CVL records a reference into or out of the fragment). A compact CVL model can be easier to maintain.

As mentioned in Section 3, CVL supports a division between the feature specification layer and the product realization layer, where the connection to the base model resides only in the product realization layer. The feature specification layer is independent of the chosen base model and does therefore not depend on the strategy for choosing a base model. However, this requires the naming policy of the feature specification layer to be independent of whether an additive or subtractive strategy is used (e.g. “No SideTrack” instead of “RemoveSideTrack”). The product realization layer will use substitutions with additive or subtractive strategies based on the kind of base model that is used.

Furthermore, in TCL the base model can either be a complete product that will be one of the products of the product line, or it can be an incomplete product that will be transformed to a valid product by CVL. Whether to use a complete product as a base model depends on the base language and on the choice of strategy for selecting a base model.

When creating the station product line we decided to follow the intermediate base model strategy, by choosing a base model that has neither maximum nor minimum set of features. Fig. 2 illustrates StationD, which we used as the base model for this product line. This station has been manually modeled in the TCL graphical editor. The reason for choosing this as a base model was to keep the base model as similar to all the product models as possible. The number of substitutions and the complexity of the substitutions can therefore be kept at a minimum.

Our experience shows that since CVL is based on substitutions on the base model, an intermediate base model which is similar to the product models can result in a simpler CVL model with fewer substitutions. However, there may in some cases be advantageous to use a maximum or a minimum base model. Using a minimum base model can ease the process of evolving the product line if new product models are required. New features can then be added to the minimum base model in a straightforward way to produce a new product model. A strategy using a maximum base model depends less on the use of library models, and may ease the maintenance of the base model and library models themselves.

4.3 CVL Library

CVL creates a product model by copying the base model and performing the selected substitutions. When a set of base model elements is replaced by another set of base model elements a copy is made of the second set of base model elements. This implies that the replacing set of base model elements can either originate in the base model itself or in another model, e.g. in a library. If the minimal base model strategy is used, some model elements, not already in the base model, may have to be added. This requires such a library where the additional model elements can be found.

A library can either consist of complete models where a set of model fragments is extracted, or it can be partial models with only the fragments themselves. In our case this library could either consist of complete stations where some model fragments are extracted (e.g. a side track), or the necessary fragments could be detached in a model.

TCL itself does not have any concepts for structuring model elements (e.g. side track). However, CVL can define fragments in the base language by recording references to and from base model elements inside the fragment. By defining types in CVL, these fragments of base model elements can be given an entity that can be configured and reused. Furthermore, these types can be given names that originate from the base language (e.g. "Additional TrackCircuit"). These fragments and types will therefore define base model elements from a set of library models that can be used in substitutions on the base model.

Our strategy for this product line was to create a dedicated fragment model, where all the necessary fragments were stored. The reason for this choice was the lack of other complete models with the fragments needed for the products. The fragment model is illustrated in the lower right of Fig. 4. This resulted in missing context for the fragments, yielding more work connecting the model elements together. Therefore, by rather selecting model elements from a library of complete stations, connecting the tracks together would have been more straightforward. However, this will require the library to either have several stations with all kinds of fragments or one complete station which contains every necessary fragment. This may not be practical since we may want to create a product with a new kind of fragment, and creating a complete new station in the library for this purpose defeats some of the intention of the product line.

We noticed that if a fragment model is used it is helpful to model the immediate context around the model elements in the fragment, to automate the process of connecting the model elements. However, this requires detailed knowledge of the base language and what kind of context that is necessary. For this reason, complete models may be a better choice for the library.

4.4 Creating the CVL Model

This section introduces the process and questions about creating the CVL model. There are two parts of the model that need to be considered: The feature specification layer and the product realization layer. These two layers can in principle be modeled by different developers since they are conceptually separate and require different level of

domain knowledge. The feature specification layer requires an overview of the product line and the products of the product line, while the product realization layer requires detailed domain knowledge of how elements can be connected and substituted.

The feature specification layer of the CVL model resembles a feature diagram. Creating an optimal feature specification layer of a CVL model requires it to be oblivious of the product realization layer (see Section 4.2). Other requirements for an optimal feature specification layer are discussed later in this section.

The product realization layer connects the feature specification layer to the base model and the substitutions. The size and nature of the fragments that are used to replace functionality of the base model have to be decided in the product realization layer. Basically including one feature, e.g. side track, can either be performed in one big operation or several small. Fragments should be created to optimize reuse such that one fragment of elements can be copied and put into several places. However, finding the right optimization can be a challenge and requires domain knowledge.

Based on the commonalities and differences in Table 1 we started creating the feature specification layer of a CVL model (i.e. a feature diagram) using the CVL graphical editor (see Fig. 4). We played around with the structure of this layer making it as flexible as possible supporting as many combination of products as possible in the beginning. This resulted in an unnecessary complex CVL product line model since every set of model element needed to be connected properly in the product realization layer.

In our final feature specification layer of the station product line model all of the stations have a base track and an additional track. Furthermore, the differences between the stations reside on one of these two tracks (i.e. between switch V2 and V1 in Fig. 2). The direction of the additional track can either be “UP” or “DOWN”, and the additional track can either have no sidetracks, one sidetrack or two sidetracks. Both the base track and additional track can also include an additional track circuit. As is specified in Table 1, StationB requires the main track to be named track 2. This results in the CVL model shown in Fig. 5. Note that to improve the readability, the product realization layer is hidden.

Several interesting issues were revealed when creating this product line model. First the question about what a good CVL model is. This question can be divided into the feature specification and product realization layers. Both layers have to be human readable to ease the selection of products and maintenance of the product line. The feature specification layer should be modeled in such a way that it is oblivious to the base model used, e.g. with right use of names. Furthermore the feature specification layer should be structured in such a way that it is easy to see the choices of products. The product realization layer should be compact regarding the number of substitutions and the complexity of the substitutions. However, if the purpose is to make a new product as quick as possible, a simple transformation using CVL with a minimal feature specification layer may be sufficient.

Another issue is the learning curve of using CVL. Using CVL to create a product line of stations requires the developer of this product line to have knowledge about CVL in addition to TCL. Even though CVL originates from a totally different domain than TCL (variability rather than train control), the CVL tool integration with the base DSL turned out to be valuable. By being able to use the base DSL editor to

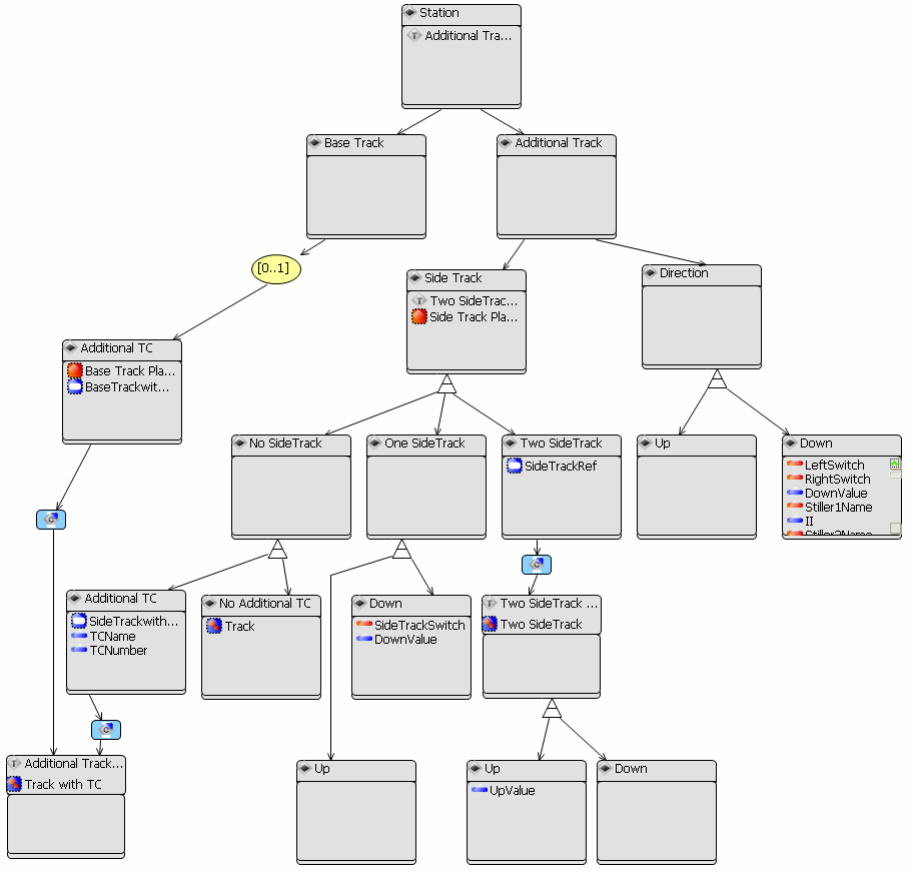


Fig. 5. Station product line in CVL

select and highlight fragments, the developer does not necessary need to know the details of CVL. Furthermore, CVL is a relatively small language with clearly defined semantics, making it less time-consuming to learn it. Experience from making the station product line shows that learning and using the simple constructs of CVL efficiently is straightforward. However, using the more advanced concepts requires more detailed knowledge of CVL.

Since CVL is a separate language to model variability, knowledge about CVL is not specific to TCL, meaning that this knowledge can be used if there is a need to develop product line models for other DSLs.

4.5 Generating Products

A CVL model does not only include the feature specification layer with information about features, but also the product realization layer with information about how to generate specific products. The resolution model can then choose which substitutions

to execute. Executing the CVL model will generate specific products (i.e. station models).

From the station product line illustrated in Fig. 5 several more than the three real target stations can be generated by carefully making another selection of features in the CVL model. For now, we are only interested in three specific products; StationB, StationD and StationS. We therefore define three resolutions to the variability model and run the CVL execution engine, which gives us the three stations mentioned. StationD is illustrated in Fig. 2, StationB in Fig. 6 and StationS in Fig. 7. Note that when the CVL model (including resolutions) is specified, the generation of the products is performed automatically. New station models are generated, and their diagrams are initialized.

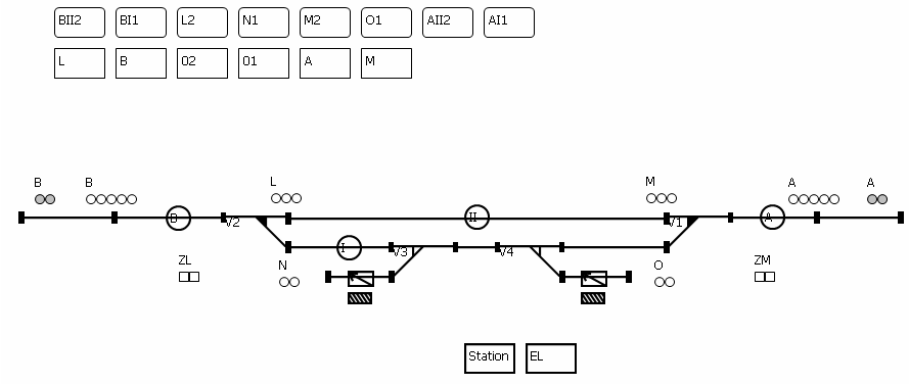


Fig. 6. StationB generated from CVL

The generated StationB has two side tracks and has direction set to "DOWN". The names of the tracks have also been changed to realize that track 2 is the main track. In the generation of StationS, two track circuits (011 and 021) have been added (i.e. rectangles representing track circuits). This can also be seen by looking at the extra line segments and endpoints on the two tracks. In addition StationS has no side tracks.

From the generated product models, the TCL code generators can be used to generate interlocking source code and interlocking tables. For each of the three products we have generated these representations and they are being validated by the signaling experts working at ABB (see Section 4.6).

Svendsen et al. [10] claim that the usage of TCL to generate interlocking source code and other representations of a station does not significantly impact the integration of safety standards or the formal techniques used to verify the correctness of the stations. The main reason for this is that TCL only automates the process of creating the representations, and that the same process of verification still can be followed. We argue that by extending TCL with CVL, we only build on top of TCL to automate the creating of stations further, and thus do not significantly impact the integration of safety standards. Each station will still run through the same kind of verification

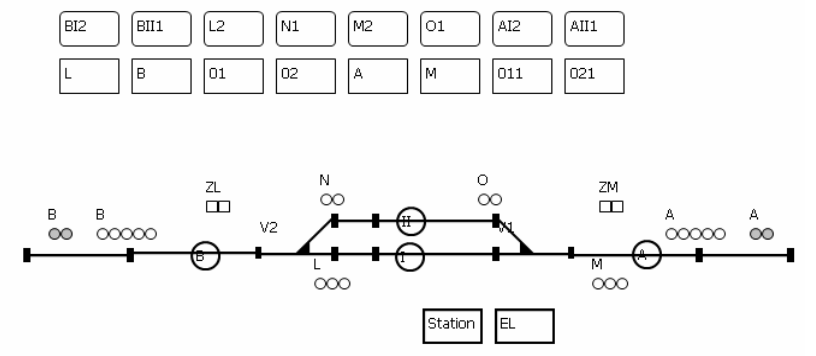


Fig. 7. StationS generated from CVL

process, keeping the safety level that was imposed by TCL. Section 4.6 explains the verification process of the stations.

4.6 Validation and Verification of the Generated Products

CVL is a generic language that can express variability in models of any DSL that is defined by a metamodel. The CVL execution will make sure that all product models that are generated comply with the metamodel of the base language. However, since CVL is generic, it is oblivious to the semantics of the base language, and can therefore create semantically wrong models. However, this can be validated using the base language tools, such as editors, code generators and model validators.

Since the train domain is a safety system, two steps are necessary for verification of the interlocking source code generated from the station products. First verification using Fagan inspection [3] has to be performed, which includes a set of rules, guidelines and checklists for use in ABB RailLock. This is first performed on the functional specification and design specification, which are checked against a predefined schema. Then it is performed once more on the interlocking source code, checking it against the functional specification and design specification. When this verification is completed, an independent party has to validate the source code against all safety requirements using a formal mathematical method. This third party is using a tool which is accepted as adequate by the Norwegian Railway authority.

Validation and verification of the interlocking source code generated from the three station products, which were produced by CVL, are currently being performed. Additional rules for checking automatically generated source code is also under development. Note that since the generated products are real stations, they have to be validated and verified using a certified process.

4.7 Summary

Fig. 8 summarizes the process of making a CVL model.

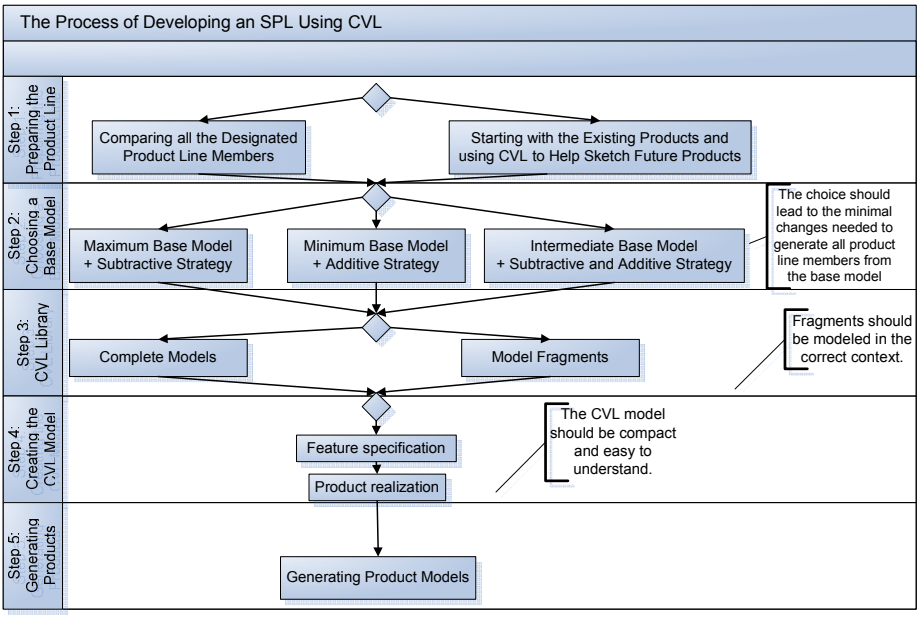


Fig. 8. The process of developing an SPL using CVL

5 Conclusions and Future Work

This paper presented a case study of CVL creating a software product line for TCL. We showed the process of creating the product line and automatically generating station product models. Furthermore we discussed the need for verification of the source code generated from these station product models. During this study we encountered some questions, where we gave some preliminary answers. As a summary, we illustrated in a figure the method to follow when creating a CVL model.

This case study was performed on three real stations with two tracks, which is the complexity of the stations being planned. CVL only records the incremental differences between base models. As long as the number of differences between base models remains stable, an increase of complexity in the base models themselves do not significantly affect the CVL approach.

This case study has shown that CVL can function as a standardized language for defining and executing a software product line to create product models. Furthermore, CVL has shown to be effective to automate the process of creating a software product line and generating its products.

Future work will include using CVL to create software product line models for other DSLs to further show the applicability of the CVL approach. How to create optimal CVL models, both on the feature specification and product realization layer, is also an issue that will be further investigated.

Other work that is also in progress is an automated approach for comparing a set of base models and deriving the CVL product line model [11]. Automating the evolution of test-cases corresponding to base models is also being investigated [9]. We believe

that such incremental analysis for safely avoiding retest of test-cases will give a huge benefit and return on investment.

Acknowledgments. The work presented here has been developed within the MoSiS project ITEA 2 – ip06035 part of the Eureka framework.

References

1. EMF, Eclipse Modeling Framework Project (Emf), http://www.eclipse.org/modeling/emf/
2. Endresen, J., Carlson, E., Moen, T., Alme, K.-J., Haugen, Ø., Olsen, G.K., Svendsen, A.: Train Control Language - Teaching Computers Interlocking. In: Computers in Railways XI (COMPRAIL 2008), Toledo, Spain (2008)
3. Fagan, M.E.: Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal 15, 182–211 (1976)
4. Fleurey, F., Haugen, Ø., Møller-Pedersen, B., Olsen, G.K., Svendsen, A., Zhang, X.: A Generic Language and Tool for Variability Modeling. SINTEF, Oslo, Norway, Technical Report SINTEF A13505 (2009)
5. Haugen, O., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding Standardized Variability to Domain Specific Languages. In: SPLC 2008, Limerick, Ireland (2008)
6. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (Foda) Feasibility Study. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. Tech. Report CMU/SEI-90-TR-21 (November 1990)
7. MOF, The Metaobject Facility Specification, http://www.omg.org/mof/
8. Oldevik, J.: Mofscript Eclipse Plug-In: Metamodel-Based Code Generation. In: Eclipse Technology Workshop (EtX) at ECOOP 2006, Nantes (2006)
9. Svendsen, A.: Application Reconfiguration Based on Variability Transformations. School of Computing, Queen's University, Kingston, Canada, Technical Report 2009-566 (2009)
10. Svendsen, A., Olsen, G.K., Endresen, J., Moen, T., Carlson, E., Alme, K.-J., Haugen, O.: The Future of Train Signaling. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 128–142. Springer, Heidelberg (2008)
11. Zhang, X.: Synthesize Software Product Line. In: The 32nd International Conference on Software Engineering, Cape Town, South Africa (2010)

Dealing with Cost Estimation in Software Product Lines: Experiences and Future Directions

Andy J. Nolan[1] and Silvia Abrahão[2]

[1] Rolls-Royce
SIN C-3, Rolls-Royce plc, PO Box 31
Derby DE24 8BJ, England
Andy.Nolan@Rolls-Royce.com
[2] ISSI Research Group, Department of Computer Science
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022, Valencia, Spain
sabrahao@dsic.upv.es

Abstract. After 5 years invested in developing accurate cost estimation tools, Rolls-Royce has learnt about the larger potential of the tools to shape many aspects of the business. A good estimation tool is a "model" of a project and is usually used to estimate cost and schedule, but it can also estimate and validate risks and opportunities. Estimation tools have unified engineering, project and business needs. The presence of good estimation tools has driven higher performance and stability in the business. It was evident we needed this capability to underpin decisions in our new Software Product Line strategy. The objective of this paper is twofold. First, we report the experiences gained in the past on the use of estimation tools. Second, we describe the current efforts and future directions on the development of an estimation tool for Software Product Lines. At the heart of the Product Line estimation tool is a simple representation of the product – represented as the number of Lines Of Code (LOC). The next generation of tool, will need to consider wider aspects of product quality in order to create more accurate estimates and support better decisions about our products.

Keywords: Cost Estimation, Software Product Lines, Industrial Experiences.

1 Introduction

The production of quality software, on time, and within budget, remains an open problem of Software Engineering that has been addressed from different approaches. An industrial approach to this problem is to use Software Product Lines (SPL). Several benefits are associated to the introduction of product lines in software development organizations such as cost reduction, time-to-market improvement, project risk reduction, and quality improvement.

However, the associated costs and the quality of the software products may greatly differ due to systematic reuse. In addition, product line engineering is often the more economical choice in the long-term run. It might not be the best choice when project managers want to amortize their core asset base across only a few products or across

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 121–135, 2010.

products with little commonality [1]. Therefore, there is a need for tools to help project managers to analyze in which situations and scenarios product line investment pays. To address this issue, several cost estimation models for Software Product Lines (SPL) have recently been proposed in the literature. However, to understand their benefits and weaknesses, it is important to analyze the experiences gathered in applying these models in industrial or organizational settings.

In this paper, we present an experience report about the use of cost estimation tools at Rolls-Royce. The objective of this paper is (i) to report the experiences gained in the past on the use of a cost estimation tool based on COCOMO (Constructive Cost Model) [3] (ii) to describe how this tool was extended for its use with software product lines as well as the lessons learned (iii) to describe future extensions for this tool based on the preliminary results obtained within the MULTIPLE (Multimodeling Approach for Quality-Aware Software Product Lines) project conducted at the Universidad Politécnica de Valencia in Spain with close collaboration of Rolls-Royce.

This paper is organized as follows. Section 2 discusses existing models and tools for cost estimation in SPL. Section 3 discusses past experiences on the use of a Cost Estimation tool at Rolls-Royce. Section 4 presents an overview of the SPL initiative launched in 2008 as well as the development of an estimation tool which was built for assessing the benefits of SPL. Section 5 describes the lessons learned and the future extensions of the tool. Section 6 presents our conclusions and further work.

2 Related Works

In the last few years several cost estimation models for software product lines have been proposed. Some representative proposals are: [16], [1], [19], [4], [7], [10], [9] and [13]. Poulin [16] proposed one of the first models for analyzing the effects of employing a systematic reuse approach. The model is based on two parameters: the relative cost of reuse (RCR) and the relative cost of writing for reuse (RCWR). The first parameter can be used for comparing the effort needed to reuse software without modification to the costs associated with developing the same software for a single use. The second parameter relates the costs of creating reusable software to the cost of creating one-time use software. These parameters can also be applied in the context of software product lines. The Poulin model uses the RCR and RCWR to calculate two other indicators (i.e., reuse cost avoidance and additional development cost) that predict savings for developing a specific project.

Böckle *et al.* [1] proposed a software product line cost model to calculate the costs and benefits that we can expect to have from various product line development situations. In particular seven reuse scenarios were identified. The cost model proposed involves the following four costs: (1) the cost to an organization of adopting the product line approach for its products; (2) the cost to develop a core asset base suited to support the product line being built; (3) the cost to develop unique software that is not based on a PL platform; (4) the cost to reuse core assets in a core asset base. The authors then analyze the cost savings of converting products to a software product line as they evolve over time.

Tomer *et al.* [19] proposed a model that enables software developers to systematically evaluate and compare alternative reuse scenarios. The model supports the clear

identification of the basic operations involved and associates a cost to each basic operation (e.g., adaptation for reuse, new for reuse, new development, cataloged asset acquisition). In 2004, Boehm *et al.* [4] proposed a software product line life cycle economics model called Constructive Product Line Investment Model (COPLIMO). The model facilitates the determination of the effects of various product line domain factors on the resulting PL returns on investment.

Since the previous cost estimation models do not properly consider the software quality cost, In *et al.* [9] proposed a quality-based product line life cycle estimation model called qCOPLIMO as an extension of the Boehm et al. model [4]. This model is based on the top of two existing models proposed as an extension of the COCOMO II model: COPLIMO, which provides a baseline cost estimation model of the SPL life cycle, and COQUALIMO which estimates the number of residual defects. The model provides a tool to estimate the effects of software quality cost for enabling cost-benefit analysis of SPL. However, quality is measured only as the cost per defect found after product release and the tool is not granular in terms of the product itself.

Clements *et al.* [7] proposed the Structured Intuitive Model for Product Line Economics (SIMPLE) model. Its purpose is to support the estimation of the costs and benefits in a product line development organization. The model suggest four basic cost functions to calculate (1) how much it costs an organization to adopt the PL approach for its products; (2) how much it costs to develop the core asset base to satisfy a given scope: (4) how much it costs to develop the unique parts of a product that are not based on assets in the core asset base; (4) how much it costs to build a product reusing core assets from a core asset base.

In [10] Lamine et *al.* introduce a new software cost estimation model for SPL called SoCoEMo-PLE. This model is based on two previous models: the integrated cost estimation model for reuse [11] and the Poulin's model [16]. The authors claim that when compared to the two costs models used, the proposed new model gives different results and presents more details because it takes into account more features of PLE development life cycle. However, no evidence for this claim was found.

Finally, other authors suggest that a decision analysis model could be integrated into the cost model to provide an interpretation to the values obtained by the cost functions. This is the case of the Nóbrega *et al.* [13] proposal where an Integrated Cost Model for Product Line Engineering (InCoME) is presented. The aim of this model is to perform investment analysis for a set of reuse scenarios to help an organization to decide if an investment in a product line is worthwhile. The model was applied in a small product line with 9 products, 10 core assets and two reuse scenarios. Although the results seem promising, the model should be applied to other organizations with larger PLs in order to test the generalizability of the results obtained.

An analysis of these cost estimation models revealed that the majority of them estimate costs and/or benefits of using SPL through several reuse scenarios (e.g., [1] [19] [13]). Other models identify a clear separation of cost estimation and investment analysis [13]. Some models consider variations in the cost of reuse [4] [9]. In general, the proposed models suggest several parametric values that must be accurately calibrated. Finally, the majority of the proposed models often not considered other factors such as quality and time-to-market.

3 Past Experiences on the Use of a Cost Estimation Tool

This section gives a brief overview of the software development and cost estimation practices at Rolls-Royce. This is important to understand as it was the foundation for building the Product Line Estimation tool shown in Section 4.

3.1 Rolls-Royce Control Systems

Rolls-Royce provides power systems and services for use on land, at sea and in the air, and operates in four global markets - civil aerospace, defense aerospace, marine and energy. In all the business sectors in which Rolls-Royce operates, there are demands for improved capability and effectiveness of the power systems, more economic and faster product development, better transition to operation (minimum post-delivery changes) and better in-service cost and availability, with commensurate reduction in cost of purchase and/or cost of ownership.

The Control Systems department of Rolls-Royce's Aerospace business is responsible for the Engine Electronic Controllers (EECs) for a range of small and large gas turbine engines, for the aerospace industry. The EEC contains a significant amount of software that is designed to 'control' the engine, as directed by the pilot, in a way that is safe for the engine, safe for the aircraft, fuel-efficient, component life efficient and environmentally efficient. We have been developing high integrity software for over 20 years and have extensive data on our processes and productivity. We have had some level of success with clone-and-own- reuse but this tended to be opportunistic from existing projects. Since 2008, we are developing our SPL for the business which has potential for both the software and hardware aspects of our engine design.

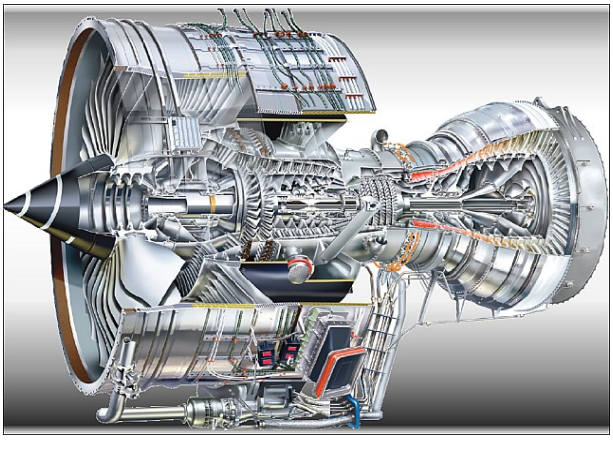


Fig. 1. Rolls-Royce Trent 900 engine used to power the A380 – the control software is in excess of 200,000 lines of code

3.2 The Adoption of COCOMO II

Since 2004, Control Systems has invested in developing reliable estimation tools to predict software development cost and schedule. The work was undertaken as part of a six-sigma Black Belt project to understand the factors that influenced good estimates. One of the outputs from that study was a calibrated estimation tool based on COCOMO II [2]. COCOMO is an algorithmic software cost estimation model developed by Boehm. The model uses a basic regression formula, with parameters that are derived from historical project data. COCOMO was first published in 1981 as a model for estimating effort, cost, and schedule for software projects.

COCOMO II is the latest extension to the original COCOMO and was developed by the combined efforts of USC-CSSE, ISR at UC Irvine, and the COCOMO II Project Affiliate Organizations. The revised cost estimation model reflects the changes in professional software development practice that have come about since the 1970s.

The process of evaluation and tool development took around 1 month of effort. The objective was to find a simple, accurate and believable estimation tool that would allow managers to express and defend the critical project assumptions in a way that the business could understand. Believable and dependable estimates were key requirements as well as having a tool that anyone could use.

It was necessary to find accurate data for historic projects and then to estimate their cost as if they were future projects. A "blind" estimate was generated and then validated against the actual project results. There was a "common cause" discrepancy which was down to tool calibration. In other cases, there were special cause exceptions which had to be investigated. At the end of the analysis, we had both a calibrated estimation tool as well as a thorough understanding of the COCOMO factors and how to drive them. This knowledge became part of the user guide and training program.

The COCOMO II model is a very simple equation relating factors to final cost and schedule. We added "front end" tools to help derive estimates for size (Lines of Code) as well as "back end" tools to help unwrap the results into resource profiles, phases, plans and even error predictions. The COCOMO model sat at the heart of an otherwise comprehensive resource/project planning tool.

We built many versions of the tool to meet the needs of different domains – including hardware development. In each case, we identified the "questions" we needed to ask about the project/business, selected the factors from COCOMO II that would address these questions, then built this into a tool. In those cases where COCOMO II could not provide the factor to address a question, we would develop our own factors, gather data from the business and perform a regression analysis to understand the sensitivity and range for the new factors. Examples of new factors included requirements volatility (from our customers) and Scrap & Rework generated from our evolutionary development approach to engine, hardware and electronic development.

3.3 What an Estimation Tool Teaches You

An important breakthrough occurred when we relished that the estimation tools, like COCOMO II, were not only useful for estimating costs, but were actually teaching us what was important about a project or product. Estimation tools are actually models of a project and like all models they are there to help you make good decisions. An estimation

tool defines a formal and objective representation of a project or business and is by definition a simplification of reality.

An estimation tool need only be as precise and accurate as required to make a meaningful and accurate decision. They are there to tell you something that you would not (or could not) know without them. They are central to good project management and control, estimation, improvement and risk management.

The estimation tools are also there to remove the subjectivity from the decision making process. It's tempting for a manager, fuelled by ego and heroics, eager to prove themselves, to exaggerate the truth or guess at key decisions. You also need a good estimation tool to help with reasoning, collaboration and negotiation in order to persuade the business to invest in the right things e.g., in a product line. We have found that a well-constructed estimate makes persuasion a whole lot easier then relying on good intentions and opinions.

3.4 The Business Benefits

Through the development and deployment of estimation tools across the business, we have seen an improvement in stability and productivity – on average around 11% cost saving per project. This is primarily because, estimation tools, like COCOMO II, are informing you of what is important. This information has shaped what we measure, how we identify and validate improvements, how we identify and mitigate risks and how we manage and estimate projects. They help us optimize and refine the business around objective reasoning rather the subjective guesswork i.e., we make better decisions.

For example, if a project is taking on novel features, or there are concerns over the aircraft maturity, we would expect a high level of requirements volatility and scrap & rework. The estimation tool would quantify the impact (increased cost and a longer program) and this would then be used to drive for changes in the development approach, risk mitigations, negotiations with the customer and so on. If this was a critical factor for success, then this attribute of the project would be carefully monitored and reported. Similarly, if in order to achieve a low cost project, you assume a high performance team, then this aspect of the project will need to be carefully monitored and reported. The output from an estimate is a measurement plan of critical factors that need to be monitored, controlled and where possible, improved.

4 The Estimation Tool for Software Product Lines

This section starts describing the software product line initiative launched at Rolls-Royce followed by the description of new estimation factors considered in the development of a cost estimation tool for SPL. The section ends by discussing lessons learned from practical experiences using the tool.

4.1 The Software Product Line Initiative

The challenges facing Rolls-Royce Control Systems are not unique; we have a program load greater than ever before. Our customers want faster development, so program timescales continue to decrease, while functionality increases and our shareholders demand lower costs and greater profitability. Each new project development

represents a significant engineering challenge, moving engineering from research to product development at a rapid pace. Even though the approach used today is competitive, our future order book growth means that we cannot sustain our current engineering approach. We need a step change in productivity. SPL will help us step up to these challenges.

In addition we have an extensive legacy portfolio that will require refresh as electronic equipment becomes obsolete. This will be an ongoing concern for any business, like Rolls-Royce, involved in long-life products, that includes electronics, and inevitably that use software solutions. It is this long-life that also has the potential for Product Line solutions to cost effectively refresh our legacy portfolio – but we have to accept that they are part of the 'future' in our initial Product Line market scope.

4.2 The Need for New Estimation Factors

When the SPL initiative was launched in 2008, the development of a reliable and comprehensive estimation tool was seen as critical to ensure we were making the right decisions. Based on some additional data from within Rolls-Royce, we developed the first version of the SPL Estimation tool [14] in under a day. Several versions later and about 4 weeks of prototypes and demos the first model was released and populated.

The original estimation tool was calibrated and developed around our existing processes and approach to software development. With the development of the product line, there were new questions we needed to ask, which led to the need for additional factors in the estimation tools.

- It needed to guide the business – the tool needs to help us persuade "the business" – it needs to be able to estimate business level cash flow and benefits and communicate trades and decisions to people who did not understand software.
- It needed to guide each project – each project should be able to use the estimation tool to make trade decisions between bespoke and Product Line assets.
- It needed to guide the architects – we need to pick the right features and the right variation mechanisms that add the greatest benefit.

In addition to these requirements, the tool needed to be able to answer the following:

- When to develop a Product line asset and when to rely on clone-and-own-reuse and bespoke development.
- The costs of development and deployment of assets for a range of variation mechanisms.
- To factor for organizational overheads and new roles not normally associated with traditional project development.
- The effect the Product Line Strategy has on the organization i.e., disruption and risk as well as the benefits from alignment of processes and objectives.
- The costs for redeveloping the Product Line architecture i.e. product line refresh.

We did not adopt COPLMO directly because it did not contain the granularity we needed or the "decision points" described above. The first estimation tool was developed and used in early 2009. The structure of the tool is shown below (see Fig. 3) and contains the following additional decision points:

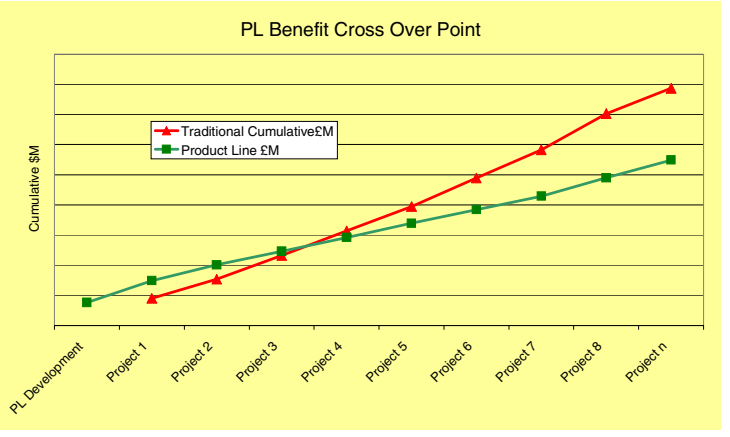


Fig. 2. The output from the Software Product Line estimation tool showing the break even point of the initiative. This information was used to persuade the business to invest in a strategic initiative rather then to focus on a project-by-project return on investment.

- Historically, size was measured in terms of the total (macro) number of lines of code. Step 1 of the SPL estimation tool provided the size of each individual asset based on a library of features from past projects.
- From historic analysis we have shown that approximately 70% of the software costs arise from 30% of assets i.e., not all assets are equal. This was not an issue for our historic estimation tools because we considered size at the macro level only. With the introduction of the SPL, we added Step 2 to understand the cost and value of each asset.
- A new step "P-Process Model" was added to model our safety-critical development process and to understand which processes are required to develop an asset and which processes are required to deploy an asset. We also recognized that the process used to develop and deploy an asset varies depending on the asset variation mechanism.
- Traditionally, features would only be considered at the time of project launch and clone-and-own approaches used to acquire assets from pat projects. Step 3 was used to map out the life of features across the future engine programs.
- Step 4 was introduced so that the SPL team could understand the costs to develop a project using the traditional methods and then compare and contrast these costs with the SPL development costs (see step 10).
- Step 5 was introduced to model the additional costs to develop a SPL asset. This information was taken from COCOMO II – the RUSE (developing for reuse) and DOCU (additional documentation) factors.
- Step 6 was used to map the deployment of SPL assets into projects to understand if there was a net benefit, per asset, when considering the development, deployment and maintenance of assets. The step also revealed those features that would be bespoke to each project.
- Step 7 was a mechanical process and generated all the information together to understand the new costs for developing a project based on SPL assets.

- Step 8 was added to model the organization costs. This was derived from the SIMPLE model [7] and consists of costs above-and-beyond traditional project development. The model contains the costing for, new organization roles and governance activities, new management activities, increased configuration management and change control, reference architecture developers, process & tool development, organizational training & orientation, consultancy costs and business level interface roles and activities.
- Step 9 was added to model the effect of introducing the SPL initiative into the business. With any change comes an initial "shock" followed by a settling in period. Also, the organization would look and behave differently. COCOMO II was again used to model the business environment.
- Step 10 performed the final analysis and compared and contrasted the benefits of the SPL with a more traditional project centric organization.

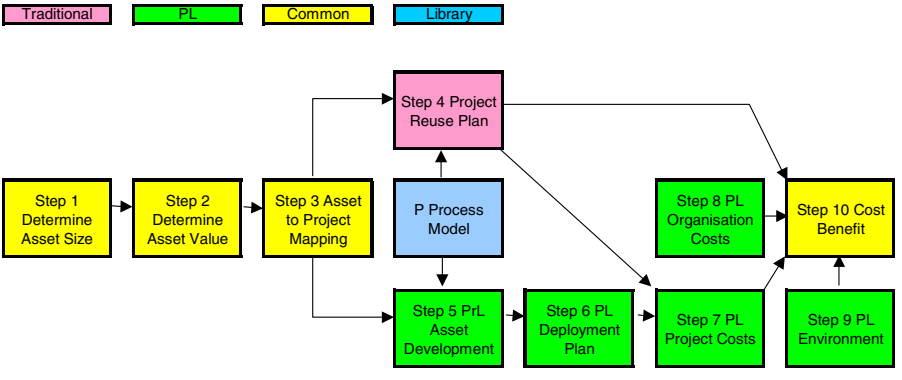


Fig. 3. The structure of the Software Product Line estimation tool

4.3 How We Use the Estimation Tool

The tool is used to perform the following activities:

- Communicate with business leaders in a language they understand. It can model the costs of investment and the point when we have return on our investments.
- Model decisions at the business, project and asset level. For example, through sequencing our projects, we can smooth out the SPL asset development program. At the asset level, the architect can make trades between variation mechanisms and net return on investment. At the project level, they can make trades between PL assets and bespoke features and use this information to either push back or negotiate with the customer.
- Perform risk analysis by understanding variation and sensitivities of decision points (modeled as factors in the estimation tool). For example, what is the effect of not having the capable development team, what is the impact of refactoring the architecture, and so on.

- Assess the return on investment of improvements. The team can perform improvement scenarios with the tool to understand which improvements give the greatest return on investment.

In terms of accuracy, after 5 years of using COCOMO II, we have an R^2 correlation of 0.98 between the predicted and actual costs i.e. at the business level, we have a accurate and normalized estimation tool. Although in reality, at the per project basis, estimates can be up to 20% in error (or higher if assumptions are wrong). We are gathering data on each asset to validate the estimation tool. As the tool was based on historic projects, adjusted for a change of process, we expect an equivalent level of accuracy.

4.4 Lessons Learned

The lessons learned from the use of the SPL estimation tool is as follows:

- The experiences at Rolls-Royce, both good and bad, could be expressed in simple meaningful terms as defined by the COCOMO II factors.
- The estimation tool could also be used to try "what if" scenarios, to elicit improvement opportunities and to validate improvement proposals.
- The tool taught the business what was important, what to manage, what to monitor, where the risks lay and where opportunities would come from.
- Despite initial reservations, the tool was calibrated and in use in only 1 month. The benefits have been on 10,000 times this effort.
- You need to have an owner who is passionate in estimation, to drive the approach into the business.
- Like any tool development, let it evolve – we had over 20 versions of the tool, each an enhancement to address new questions that came to light during use.
- We had to make a decision between developing a simple tool that anyone could use and a tool for experts – we opted for an experts tool making both development and deployment far easier and allowing greater freedom to add complex decision points into the tool designed around the use base.
- Never underestimate the drag as people would rather rely on subjectivity rather than objective reasoning but never underestimate the power of a well constructed, formulated and reasoned argument.
- The tool has been successful mainly because it can meet the needs of a wide range of users e.g., architects, project leaders and business leaders.

We also observed that future versions of the tool can be adopted to provide different view points for different business needs. At present, we can model cash flow but future tools can more accurately model schedule, asset availability, etc. Other views can be added to represent the need for key resources and resource planning.

5 Future Directions

The SPL estimation tool can answer many questions about the development environment and trades between architectural, project and business decisions, but contains only

an approximation of the product itself. At present, the product is represented only as lines of code, adjusted for complexity or difficulty. The next generation of estimation tools will need to consider, in greater detail, *quality attributes* about the assets we are developing. In this sense, we are working currently on extensions for this tool based on the preliminary results obtained within the MULTIPLE project conducted at the Universidad Politécnica de Valencia, in Spain, with close collaboration of Rolls Royce. In this section, we give a brief overview about this project and discuss how the results obtained on it can be transferred to Rolls-Royce.

5.1 The MULTIPLE Project

MULTIPLE (Multimodeling Approach for Quality-Aware Software Product Lines) is a three-year project (2010-2013) supported by the Spanish Ministry of Science and Innovation. Rolls-Royce participates as an EPO entity – declaring interest for assessing the benefits of the results derived from the project and possibly exploit them.

The objective of this project is to define and implement a technological framework for developing high-quality SPL. This framework is based on the existence of several models or system views (e.g., functionality, features, quality, cost) with relationships among them. This approach implies the parameterization of the software production process by means of a Multimodel which is able to capture the different views of the product and the relationships among them.

As part of this project, we developed a Quality Model for conducting the production plan of a SPL [12]. It is one of the views of the Multimodel and captures the quality attributes relevant for the domain, the quality attributes relevant to certain products of the family, as well as the variability among these attributes. The model allows measuring the properties of several artifacts of a SPL (e.g., core asset base, core assets, SPL architecture, product architecture) by providing quality metrics and trade-off analysis mechanisms in order to help architects to select core assets that meet the business need.

The next section describes our current efforts on using the quality attributes of the quality model as new parameters to the cost estimation tool.

5.2 Extension Mechanisms for the Rolls-Royce Cost Estimation Tool for SPL

Cost is usually an important factor in reuse-based development. However, other factors, such as product quality and time-to-market, are also expected to improve by reusing software assets. Lower error rates, higher maturity products and more complete functionality all contribute to improved customer satisfaction. Currently, we are extending the cost model for SPL with two additional factors: quality and variability.

5.2.1 Quality

To define the relevant quality attributes to be used as new parameters in the cost estimation model for SPL, we took the complete SQuaRE-based Quality Model developed for the MULTIPLE project [12] and discussed the relative importance of each quality attribute to the safety-critical embedded systems domain and SPL. Each quality attribute was classified according to the following scale: High (mandatory for this domain), Average (important but not mandatory) or Low (not important for this domain). An extract of the results is presented in Appendix A. In general, the results

show that the *Operability* quality characteristic and its associated quality attributes are not relevant for safety-critical embedded systems since it assess the degree to which the software product can be understood, learned, used and attractive to the user. The most relevant characteristics and quality attributes is as follows:

- *Functional suitability*: the degree to which the software product provides functions that meet stated and implied needs when the software is used under specified conditions. We need to develop assets that have met the appropriate certification standards, are functionally correct, are accurate and have deterministic behaviour.
- *Reliability*: the degree to which the software product can maintain a specified level of performance when used under specified conditions. Performance, in our case, can include fault tolerance, error recovery, coping with the loss of engine signals, hardware failure conditions and so on.
- *Performance efficiency*: the degree to which the software product provides appropriate performance, relative to the amount of resources used, under stated conditions. Performance is measured in terms as timing and memory utilisation as well as response times to stimulus from the aircraft or engine.
- *Compatibility & transferability*: The ability of two or more software components to exchange information and/or to perform their required functions while sharing the same hardware or software environment. We need to understand the level of compatibility with the electronic hardware standards, engine configurations, and airframe communication.
- *Maintainability*: the degree to which the software product can be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications. This issue is not limited to just safety-critical software but the effects of change can be disproportionately high in the safety-critical domain because of the need to capture the certification evidence.

These attributes have a major impact on the cost of a SPL for safety-critical embedded systems. In addition, we identified new attributes for *Safety* and *Affordability* that should be incorporated to the Quality Model. The attributes for safety are as follows: Predictability (the degree to which the software behaviour is predictable), Completeness (the degree of test coverage for behaviour), Compliance (the degree of compliance to industry safety standards), and Protection (degree of protection against unsafe conditions). The quality attributes for affordability includes the cost to develop, cost to maintain and cost to deploy in new situation/context.

Future works includes the definition of quality attribute scenarios following an approach similar to the one performed by the SEI for eliciting and representing quality attribute requirements observed in practice. In [15], a distribution of quality attribute concerns according to the SEI-led ATAM evaluations is presented. The top 3 out of 140 attributes are: Modifiability (concern = New/revised functionality/ components; distribution = 6.4%), Usability (concern = Operability; distribution = 4.1%) and Modifiability (concern = Upgrade/add hardware components; distribution = 3.9%).

5.2.2 Variability

Variability may add customer value but allowing too much variability could lead to substantial follow-up costs during the lifecycle. Therefore, variability should be added as a cost driver during the whole SPL lifecycle since each feature must be maintained integrated in subsequent releases, tested and possibly considered during deployment and customer support. In addition, we should also take into account the variability that may exist in the different products of the family with respect to the quality attributes. Therefore, two dimensions of variability should be considered: (1) for the whole SPL lifecycle and (2) with respect to the quality attributes. In the safety-critical domain, it will be necessary to prove the product is safe for all variants and combinations of variability. Too much variability could have an exponential impact on the verification, validation and certification activities.

5.2.3 Other Extension Mechanisms

New mechanisms should be defined and used in conjunction to the cost estimation model for SPL to provide relevant information about cost-benefits to the business. For instance, a mechanism to relate each business goal with the quality attributes. In addition, adding or removing functional features may influence the quality of the product family. Therefore, we need a new mechanism to relate each feature with respect to the quality attributes. There is also a need for other mechanisms to analyze the impacts among quality attributes and to identify potential conflicts among them.

6 Closing Remarks

Along with each generation of the estimation tool came new questions to be answered. Each new generation of the tool then added new factors (or factorization) to help answer those questions. As we refined our understanding and fidelity of management, we needed a higher fidelity of factors in the estimation tool.

The first generation of the estimation tools focused heavily on the development environment, being able to understand, accommodate, monitor, control and improve aspects of the development environment. The first generation estimation tools considered the software product as a single large entity and defined it in terms of lines of code. The second major generation of the tool was designed to answer new questions about the product line and to do this, we needed a refined understanding of the software architecture, its functional breakdown, variation mechanisms and the costs to develop and deploy assets.

The software Product Line estimation tool can answer many questions about the development environment and trades between architectural, project and business decisions, but contains only an approximation of the product itself. At present, the product is represented only as lines of code per feature, adjusted for complexity or difficulty. The next generation of estimation tools will need to consider, in greater detail, quality attributes about the assets we are developing. We also need to further empirically validate the accuracy of the estimates obtained using the SPL estimation tool.

Acknowledgments. This research is supported by the MULTIPLE project (with ref. TIN2009-13838) funded by the “Ministerio de Ciencia e Innovación (Spain)”.

References

1. Böckle, G., Clements, P., McGregor, J.D., Muthig, D., Schmid, K.: Calculating ROI for Software Product Lines. IEEE Software (May/June 2004)
2. Boehm, B., Abts, C., Brown, A.W., Chulani, S., Clark, B.K., Horowitz, E., Madachy, R., Reifer, D., Steece, B.: Software Cost Estimation with COCOMO II. Prentice-Hall, Englewood Cliffs (2000)
3. Boehm, B.: Software engineering economics. Prentice-Hall, Englewood Cliffs (1981)
4. Boehm, B., Brown, A.W., Madachy, R., Yang, Y.: A Software Product Line Life Cycle Cost Estimation Model. In: Proceedings of the International Symposium on Empirical Software Engineering (ISESE 2004), pp. 156–164 (2004)
5. Chen, Y., Gannod, G.C., Collofello, J.S.: Software Product Line Process Simulator. In: 6th Int. Workshop on Software Process Simulation and Modeling (May 2005)
6. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston (2001)
7. Clements, P., McGregor, J.D., Cohen, S.G.: The Structured Intuitive Model for Product Line Economics (SIMPLE), CMU/SEI-2005-TR-003 (2005)
8. Cohen, S.: Predicting When Product Line Investment Pays, Technical Note, CMU/SEI-2003-TN-017 (2003)
9. In, H.P., Baik, J., Kim, S., Yang, Y., Boehm, B.: A Quality-Based Cost Estimation Model for the Product Line Life Cycle. Communications of the ACM 49(12) (December 2006)
10. Lamine, S.B.A.B., Jilani, L.L., Ghezala, H.H.B.: A Software Cost Estimation Model for a Product Line Engineering Approach: Supporting tool and UML Modeling. In: 3rd ACIS Int. Conf. on Software Engineering Research, Management and Applications (2005)
11. Mili, A., Chmiel, S.F., Gottumukkala, R., Zhang, L.: An integrated cost model for software reuse. In: Proc. of the 22nd International Conference on Software Engineering, Limerick, Ireland, pp. 157–166. ACM Press, New York (2000)
12. Montagud, S.: A SQuaRE-based Quality Evaluation Method for Software Product Lines, MSc. Thesis, PhD Program on Software Engineering, Formal Methods and Information Systems, Dept. of Computer Science, Universidad Politécncia de Valencia, Dic. (2009)
13. Nóbrega, J.P., Almeida, E.S., Meira, S.: InCoME: Integrated Cost Model for Product Line Engineering. In: Proceedings of the 34th Euromicro Conference Software Engineering and Advanced Applications (SEAA 2008), pp. 27–34 (2008)
14. Nolan, A.J.: Building a Comprehensive Software Product Line Cost Model. In: McGregor, J.D., Muthing, D. (eds.) Proceedings of the 13th International Software Product Line Conference (SPLC 2009), San Francisco-CA, USA. IEEE Press, Los Alamitos (2009)
15. Ozkaya, I., Bass, L., Nord, R.L., Sangwan, R.S.: Making Practical Use of Quality Attribute Information. IEEE Software, 25–33 (March/April 2008)
16. Poulin, J.S.: The Economics of Product Line Development (1997), `http://home.stny.rr.com/jeffreypoulin/Papers/IJAST97/ijast97.html`
17. Schackmann, H., Lichter, H.: International Workshop on Software Product Management (IWSPM 2006 - RE 2006 Workshop), A Cost-Based Approach to Software Product Line Management. Minneapolis/St. Paul, Minnesota (2006)
18. Schmid, K.: An Initial Model of Product Line Economics. In: Proceedings of the 4th International Workshop on Product Family Engineering (PFE-4), pp. 38–50 (2001)
19. Tomer, A., Goldin, L., Kuflik, T., Kimchi, E., Schach, S.R.: Evaluating Software Reuse Alternatives: A Model and Its Application to an Industrial Case Study. IEEE Transactions on Software Engineering 30(9) (September 2004)

Appendix A

Table A-1. Excerpt of the Quality Model showing the relative importance for quality attributes

Quality Charac.	Attribute	Description	What is probably today	What it should be for SPL
Functional suitability	The degree to which the product provides functions that meet stated and implied needs		Relative Importance	
	Appropriateness	The degree to which the software product provides an appropriate set of functions for specified tasks and user objectives.	[High] Absolutely Mandatory for Safety and business needs	[High] Absolutely Mandatory for Safety and business needs
	Functional suitability compliance	The degree to which the product adheres to standards, conventions or regulations in laws and similar prescriptions relating to functional suitability.	[High] Mandatory compliance to Do-178B	[High] Mandatory compliance to Do-178B
Reliability	The degree to which the software product can maintain a specified level of performance when used under specified conditions		Relative Importance	
	Availability	The degree to which a software component is operational and available when required for use.	[Medium] Obviously we want this but we can mitigate	[High] Otherwise SPL fails
	Fault tolerance	The degree to which the product can maintain a specified level of performance in cases of software faults or of infringement of its specified interface.	[High] We design this into the architecture and coding/design standards	[High] We design this into the architecture and coding/design standards
Maintainability	The degree to which the software product can be modified		Relative Importance	
	Modularity	The degree to which a system is composed of discrete components such that a change to one component has minimal impact on other components.	[High] Driven through the architecture	[High] Driven through the architecture
	Reusability	The degree to which an asset can be used in more than one software system, or in building other assets.	[Medium] We are not good at this	[High]

Evolution of the Linux Kernel Variability Model

Rafael Lotufo[1], Steven She[1], Thorsten Berger[2],
Krzysztof Czarnecki[1], and Andrzej Wąsowski[3]

[1] University of Waterloo, Ontario
{rlotufo,kczarnec,shshe}@gsd.uwaterloo.ca
[2] University of Leipzig, Germany
berger@informatik.uni-leipzig.de
[3] IT University of Copenhagen, Denmark
wasowski@itu.dk

Abstract. Understanding the challenges faced by real projects in evolving variability models, is a prerequisite for providing adequate support for such undertakings. We study the evolution of a model describing features and configurations in a large product line—the Linux kernel variability model. We analyze this evolution quantitatively and qualitatively.

Our primary finding is that the Linux kernel model appears to evolve surprisingly smoothly. In the analyzed period, the number of features had doubled, and still the structural complexity of the model remained roughly the same. Furthermore, we provide an in-depth look at the effect of the kernel's development methodologies on the evolution of its model. We also include evidence about edit operations applied in practice, evidence of challenges in maintaining large models, and a range of recommendations (and open problems) for builders of modeling tools.

1 Introduction

The cost of variability management in software product lines is meant to be offset by the savings in deployment of product variants over time. Product families with long lifetime and large number of variants should provide a bigger return over time. For these reasons a product line architecture is typically implemented in large projects with a long time horizon. The time horizon and the sheer size of these projects place coping with *scale* and *evolution* as the forefront challenges in successfully running software product lines.

Variability models evolve and grow together with the evolution and growth of the product line itself. Thus realistic feature models are large and complex [1], reflecting the scale of growth and evolution. Nevertheless, evolution of real variability models has not been studied. Multiple authors have been interested in reasoning about feature model editing [2,3], in semantics of feature model refactorings [4], or in synchronizing artifacts in product lines [5,6], which indeed, as we shall see, is a major challenge in maintaining a variability model. However, none of these works was driven by documented challenges faced by practitioners.

We set out to study how feature models evolve, and the main challenges encountered in the process. Do the cross-tree constraints deteriorate or dominate

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 136–150, 2010.

hierarchy over time? Does the number of cross-tree dependencies become unmanageable? Is the model evolved ahead of the source code, along with the source code, or following the source code? We address these and similar questions, hoping to inspire researchers and industries invested in building tools and analysis techniques for variability modeling.

The subject of our study is the Linux feature model. As argued previously [1], the model extracted from Linux Kconfig is, so far, the largest feature model publicly known and freely available. We study the evolution of this model over the last five years, when Linux and the model were already at a mature stage. The model demonstrates that a lasting evolution of a huge product family is feasible and does not necessarily deteriorate the quality of the feature model. Despite the number of features doubling in the studied period, structural and semantic properties of the model have changed only slightly over time, retaining the desirable aspects, such as balanced composition and limited feature interaction.

The main contributions of this work are the following:

- A study of evolution of a real-world, large and mature variability model;
- Evidence of what operations on feature models are performed in practice;
- Evidence of what refactorings are applied to models in practice;
- Evidence of the difficulty for humans to reason about feature constraints;
- Input for designers of tools and techniques supporting model evolution.

We give background on Linux and its configuration language in Section 2. Section 3 justifies the choice of the experiment subject and period, and sketches the experiment design. Section 4 presents and analyzes the collected data. Remaining sections summarize threats to validity, related work, and our conclusions.

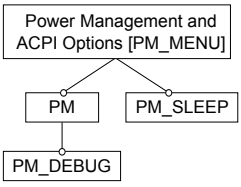
2 The Linux Kernel and Its Variability Model

Born in 1991, the Linux kernel is one of the most mature open source projects as of writing, and continues to be rapidly developed. It remains a crucial component of numerous open and closed source projects, including distributions of the GNU/Linux operating system, mobile phones, netbook computers, network routers, hardware media players and similar appliances. This diversity of applications and users, enforces a highly configurable architecture on the kernel. Indeed Linux kernel is among the largest well documented software product lines studied so far [1,7].

Linux development community comprises both volunteers and paid developers recruiting from more than 200 companies including Red Hat, IBM, Intel, Oracle, Google and Microsoft among others [8]. The maturity of the project manifests in multiple metrics such as the codebase size (exceeding 8 million lines), the number of active developers (600–1200 per release and growing), and the level of activity (up to 10000 patches per release).

Kernel versions numbers are triples: triple 2.6.12 represents a kernel from the 2.6 branch at minor revision number 12. A new minor revision is released every 3 months. All revisions studied in this paper belong to the 2.6 branch, and thus

```
1  menu "Power management and ACPI options"
2    depends on !X86_VOYAGER
3    config PM
4      bool "Power Management support"
5      depends on !IA64_HP_SIM
6    config PM_DEBUG
7      bool "Power Management Debug Support"
8      depends on PM
9    config PM_SLEEP
10     bool
11     depends on SUSPEND || HIBERNATION
12                || XEN_SAVE_RESTORE
13     default y
14 endmenu
```



$PM\_MENU \rightarrow \neg$ X86_VOYAGER
$PM \rightarrow \neg$ IA64_HP_SIM
$PM\_SLEEP \leftrightarrow$
SUSPEND || HIBERNATION ||
XEN_SAVE_RESTORE

Fig. 1. A simple Kconfig model (left) and the corresponding feature model (right)

we will only use minor revision numbers when referring to them (so 12 denotes 2.6.12). We shall use the terms revision, release and version interchangeably.

The Linux kernel contains an explicit feature model (the Linux kernel feature model) expressed in the domain specific language called Kconfig. The Kconfig language was officially merged into revision 2.5.45 in October 2002 [9]. It has been the language for the Linux kernel feature model ever since. Thus, the Linux kernel feature model is a mature model with as much as 8 years of history in its current form (and a good prehistory in predecessor specification languages). We shall analyze the last five years of this history, which span the mature stage of the model evolution, still characterized by an unprecedented growth.

We now present the Kconfig language. Configuration options are known as *configs* in Kconfig. They can be nested under other configs and grouped under *menus*, *menuconfigs* and *choice groups*. The kernel configurator renders the model as a tree of options, which users select to specify the configuration to be built.

Figure 1 shows a fragment of the Linux variability model, containing a menu (line 1) with two Boolean configs as children: PM (lines 3–5) and PM_SLEEP (lines 9–13). Configs are named parameters with a specified type. A boolean config is a choice between presence and absence. All configs in Figure 1 are bool (e.g. line 4). Integer configs specify options such as buffer sizes. String configs specify names of, for example, files or disk partitions. Integer and string configs are *entry-field configs*—shown as editable fields in the configuration tool.

A *depends-on* clause introduces a hard dependency. For example, PM can only be selected if IA64_HP_SIM is not (line 5). Conversely, a *select* clause (not shown) enforces immediate selection of another config when this config is selected by the user. Nesting is inferred by feature ordering and dependency: for example PM_DEBUG is nested under PM (line 8). A *default* clause sets an initial value, which can be overridden by the user. For example, PM_SLEEP defaults to y.

Menus are not optional and are used for grouping, like mandatory non-leaf features in feature models. *Choices* (not shown) group configs, which we call

choice configs, into alternatives—effectively allowing modeling of XOR and OR groups. *Menuconfigs* are menus that can be selected, typically used to enable and disable all descendant configs.

As in [1,9], we interpret the hierarchy of configs, menuconfigs, menus, and choices as the *Linux feature model*. The right part of Figure 1 shows the feature model for the Kconfig example in the left part of the figure. Table 1 maps basic Kconfig concepts to feature modeling concepts. An entry-field config maps to a mandatory feature with an attribute of an appropriate type, integer or string. Conditional menus map to optional features; unconditional menus to mandatory features. We map a choice to a feature with a group containing the choice configs. A mandatory (optional) choice maps to a mandatory (optional) feature with an XOR-group. More details on Kconfig and its interpretation as a feature model are available in [1].

3 The Experiment

3.1 Linux Feature Model as a Subject

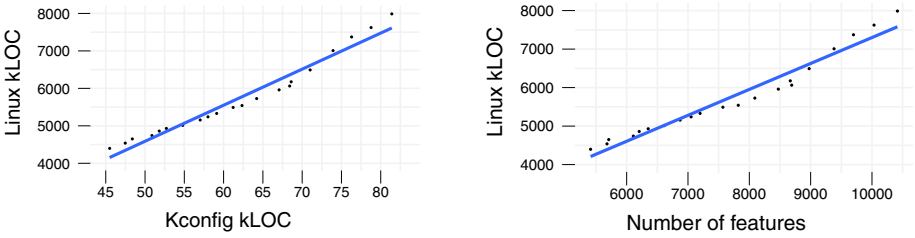
Before we proceed to our experiment, let us address the basic relevance: are the Linux variability model and the selected period of evolution relevant to study?

We analyze the evolution of the Linux kernel feature model between revisions 12 and 32—a period extending over almost 5 years, in which the Linux code base was already large and well established, while still growing rapidly. Meanwhile, the size of the kernel, measured as the number of lines, has doubled. It was also a period of intensive changes to the Kconfig model, since maintenance and evolution of this model follows the source code closely in size and in time.

Figure 2 plots the size of the Linux source code against the size of the Kconfig files (a), and against the number of features declared in Kconfig (b). Since all these measures are growing monotonically with time, the progression of samples from the origin towards the right top corner is ordered by revision numbers. Each point represents one of the 21 revisions between 12 and 32. We observe that in

Table 1. A simplified mapping of Kconfig models to feature models [1]

Kconfig concepts		**Feature modeling concepts**
Boolean config		Optional feature
Entry-field config		Mandatory feature
Conditional menu		Optional feature
Unconditional menu		Mandatory feature
Mandatory Choice		Mandatory feature + (XOR,OR)-group
Optional Choice	⇒	Optional feature + (XOR,OR)-group
Config, menu or choice nesting	⇒	Sub-feature relation
Visibility conditions, Selects, Constraining defaults	⇒	Cross-tree constraint



(a) Total kLOC for Linux against kLOC of Kconfig files

(b) Total kLOC for Linux against the number of features in Kconfig files

Fig. 2. Evolution of number of features and lines of code from revisions 12 to 32

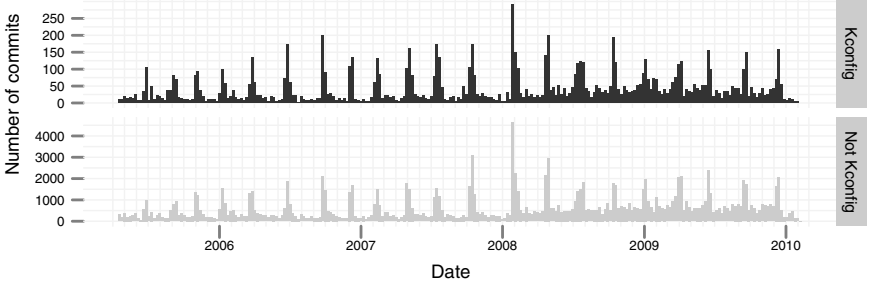


Fig. 3. Number of commits per week that touch Kconfig files compared to number of commits that do not. Each spike matches one of the 21 revisions analyzed.

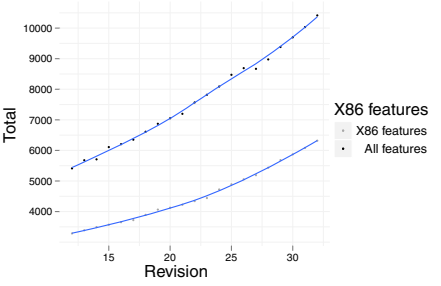
the given period the feature model grows almost linearly with the amount of source code, and its textual representation (Kconfig files).

Figure 3 shows the number of patches added weekly to the Linux source code that modify, and also that do not modify, Kconfig files. Both numbers exhibit almost identical 'heart-beat' patterns, suggesting a causal dependency between changes to the model and to the code.

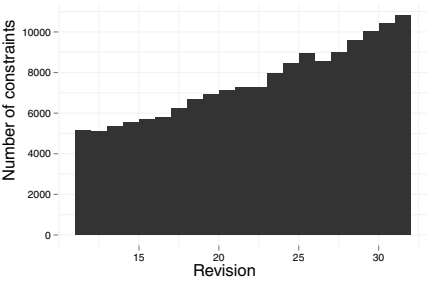
All three diagrams are strong quantitative indications that the development of the Linux kernel is feature-driven, since the source code is modified and grows along with the modifications to and growth of the feature model. This feature-oriented development on large scale makes the Linux model an interesting and relevant subject of investigation. We can expect that challenges faced by Linux maintainers can be exemplary also for other projects of similar maturity.

To scope our investigation, we focus on the model for the x86 architecture, extracted from the main line of development[1]. This scoping does not significantly skew our results, since x86 is the longest supported, the largest and the most widespread architecture of the kernel. We have verified that the x86 feature model exhibits the same pattern of growth as the entire model. For example, see the growth of the number of features in Figure 4a plotted for the entire kernel, and for the x86 architecture. Note that the x86 architecture was created

[1] git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git



(a) Total number of features in Linux Kconfig and x86 architecture



(b) # of constraints per revision

Fig. 4. Growth in number of features and number of constraints

in release 24 by merging the 32-bit i386 and the 64-bit x86_64 architectures.[2] From releases 12 to 23 we consider the i386 architecture as the x86 architecture.

3.2 Data Acquisition

Since release 2.6.12 the Linux kernel uses Git (http://git.or.cz) as its version control system. The Git commit history is a series of atomic patches extending over multiple files, each of which contains the commit log, and a detailed explanation of the patch. In the Linux project each patch is reviewed and signed-off by several experts. Since Git allows history rewriting [10], only few patches contain incorrect or misleading information. Thus we consider the Linux Git repository a trustworthy source of information and we limit our attention span to releases in which Git was used. Although historical revisions predating 2.6.12 have been converted to the Git format, one has to keep in mind that they were created using different tools, and thus in different circumstances. To assure quality and consistency of our mining, we chose not to extend our investigation before 2.6.12.

We use a parser [1] extracted from the Linux xconfig configurator to build the feature hierarchy tree. This ensures reliable and consistent interpretation of syntax. We use CLOC (http://cloc.sourceforge.net) to measure code size. Blank lines, comments, and files not recognized as code by CLOC are ignored.

4 Evolution of the Linux Kernel Variability Model

We shall now present and analyze the collected data, dividing it into two parts: Section 4.1 on the macro-scale, and Section 4.2 on the micro-scale.

4.1 Evolution of Model Characteristics

In [1] we have identified and described a number of characteristics of the Linux kernel feature model. We will now analyze how they change over time.

[2] More details at http://kernelnewbies.org/Linux_2_6_24

Model Size. As previously said (see Figure 4a), the number of features of the x86 feature model has almost doubled during the studied period, growing from 3284 in release 12 to 6319 in 32. The growth is steady and uniformly distributed, indicating a regular development pace, and a repetitive development cycle. Also, as seen in Figure 2 this growth is paralleled by the code growth, with a roughly constant feature granularity (measured as average SLOC per feature).

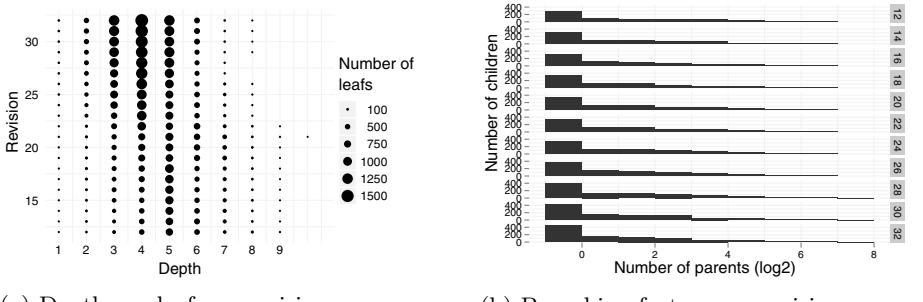
Depth of Leaves. As the hierarchy is the only structuring construct in feature models (and in Kconfig), the growth of the model must necessarily influence either depth or breadth of the hierarchy.

In revision 12 the deepest leaf is at depth 9, and most leaves are at depth 5. Somewhat counter-intuitively, both maximum depth and dominant depth have decreased over time: maximum depth in revision 32 is 8 and most leafs are now at a depth of 4—see Figure 5a. The dominant depth has decreased, even as the number of features at depth 5 continued to increase. So this feature model has been growing in width, not in height.

As reported in [1], the Linux kernel feature model is shallow and has been so at least since revision 12. The Linux project evolves the model in such a way that the basic structure of the hierarchy remains stable over considerable periods of time, despite massive changes to features themselves.

Constraints. In [1] we report that the Linux kernel feature model constraints are mostly of type 'requires'. However we did find considerably many constraints involving more than one feature, with extreme cases of constraints containing up to 22 features. We now examine how these properties change over time.

Figure 4b shows that the number of constraints has increased over time: the amount of constraints in revision 32 is almost double that of revision 12. It is interesting to note that contrary to the belief that the number of constraints grows quadratically with the number of features, the two numbers have grown in the same proportion in the given period. Again, the Linux kernel model demonstrates that it is feasible to construct software architectures and models that only induce constant number of dependencies per feature. In this sense it proves that feature models are a feasible modeling language for large projects.



(a) Depth per leaf per revision

(b) Branching factor, per revision

Fig. 5. Basic characteristics of hierarchy and branching factor across revisions 12–32

Branching Factor. We also measured the branching factor for each of the revisions, in the same manner as in [1] (see Fig. 5b). We found that there was no significant change in the shape of the histogram of children per feature, except that the number of features for each branching factor have increased. For example, there were approximately 300 features with one child, and outliers with 120 children in revision 12. In revision 32 these numbers are 400 and 160.

4.2 Summary of Model Content Changes

We have seen that the feature model has undergone many changes between revision 12 and 32. In particular, the size, average depth and number of constraints were affected. We shall now look deeper into these changes. We will characterize the edits that affected these characteristics, their overall motivation, and the implications for tool developers. For the purpose of this investigation, we define an *edit* to a feature model as a series of changes committed in the same patch.

In order to analyze motivation for individual edits to the model, we have selected a set of 200 uniformly random patches from the Git log, out of 8726, that in the given period touch Kconfig files. We have used this sample for training, to identify six categories of reasons for changes in the Linux model:

New functionality: model modifications when adding new configurable functionality;
Retiring obsolete features: modifications removing functionality from the project.
Clean-up/maintainability: modifications that aim at improving usability and maintenance of the feature model;
Adherence to changes in C code: model modifications reflecting changes made to dependencies in C code in the same patch;
Build fix: reactive modifications that adjust the feature model to reflect changed dependencies in C code in prior patches;
Change variability: adjustments to the set of legal configurations of the feature model without adding code for new functionality.

After defining the above categories, we have independently selected another 200 patches, but this time out of 7384 of those touching Kconfig files used in the x86 architecture model. We classified this sample manually and interpreted the results. We have restricted ourselves to the x86 features, in order to be able to relate the results of this study to characteristics computed for x86 in Section 4.1.

Figure 6 shows the results. In the following paragraphs we discuss each of the categories in detail, outlining its typical edit patterns, the effects it has on the feature model, and the tooling that would be desirable for the given scenario.

New functionality. Close to half of the patches in our sample were related to functionality changes—either adding to, or removing from the kernel. Each of these predominantly simple patches comprises of adding functional C code, updating a Makefile to specify how the new code will be compiled into the kernel, and typically adding one new config with simple constraints and a documenting

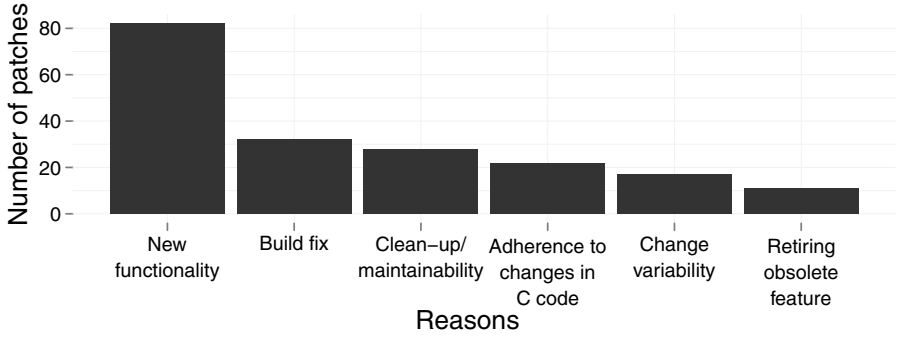


Fig. 6. Reason for edits (sample)

help text. When more than one feature is added, they are typically siblings. Most often these operations do not add further constraints to the model.

Feature additions rarely make intrusive changes to the feature model hierarchy, as almost 87% of all new features are added at leaves. Details are available in Figure 7a, which also shows that more than 50% of new features are added as leaves at levels 3, 4 and 5. Our hypothesis is that this is because the x86 architecture is very mature, and developers add features to existing elements ("slots") of the architecture, without extending the architecture itself.

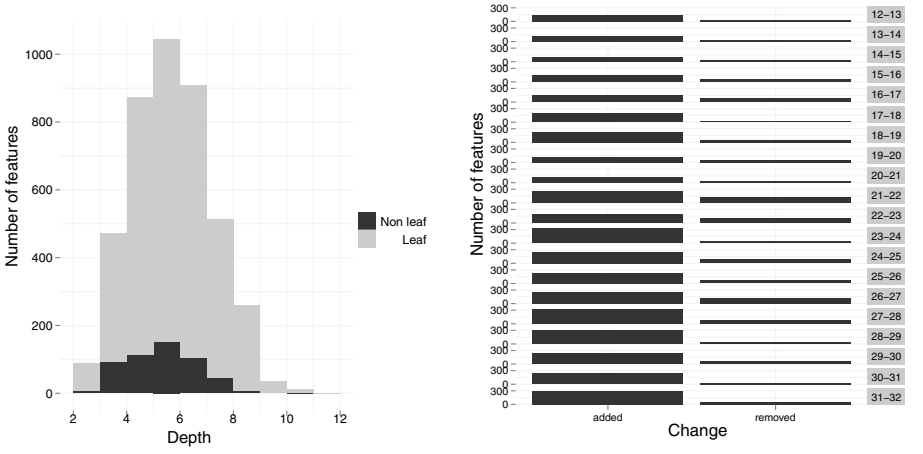
Figure 7b shows the number of features added and removed in consecutive releases. As expected, the number of feature additions in total and per release exceeds that of feature removals. Figure 7b also shows that the number of features added between releases 12 to 23 is much smaller than the additions from release 24 to 32, and correlates well with Figure 3, where we see much higher numbers of commits per week after revision 24 (January 2008).

Thus, most edits to the Linux kernel between revision 12 and 32 add new functionality, new drivers and features, as opposed to performing code and model refactorings. This is consistent with our findings (Figure 6) that almost half of the sampled patches are motivated by inclusion of new functionality and features. This also correlates to the findings of [11] which found that the super-linear growth of the Linux kernel from 1994 to 2001 was due to the growth of driver code, where drivers are typically added as new features.

Retiring obsolete features. As previously shown (Figures 4a, 6 and 7b), removing features is a rare motive for edits. We have found that this mostly happens when features are no longer supported by any developer, or when the feature has been replaced by another, making the former feature obsolete. These operations are the inverse of the operations shown in the previous section, and mostly consist of removing C code, build instructions and the related config from the model.

Notably, the kernel project maintains a formal schedule of retiring features and code, which can be found in the project tree.[3] Every entry in this file describes

[3] The file is `Documentation/feature-removal-schedule.txt`



(a) Depth of added features

(b) # of added, removed and moved features

Fig. 7. Added and removed features for releases 12–32

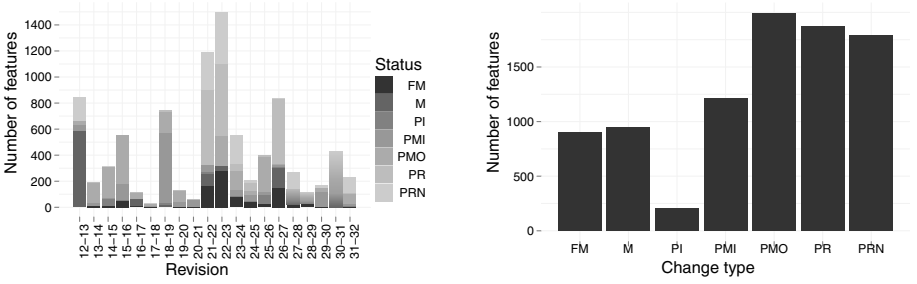
what exactly is removed, why it is happening, and who is performing the removal. This was the first time when the authors of this paper experienced such a formalized and feature-driven (!) process for phasing out code.

Retiring features is not well supported by existing tools and model manipulation techniques. It would be desirable to provide tools that: (a) eliminate features from the model (including from the cross-tree constraints, without affecting the configuration space of the other features, and possibly performing diagnosis about the impact of removal), and (b) use traceability links to find code related to the feature to verify correctness of code retiring.

Clean-up/maintainability. As seen in Fig. 6, developers frequently edit the model to improve its maintenance and usability. These edits typically focus on the end users by improving help text and feature descriptions or by refactoring the hierarchy. Constraint refactorings are a common consequence of hierarchy refactoring in the Kconfig syntax.

We assume that a feature f was subject to hierarchy refactoring if its parent has changed between releases. Let f be a feature that moved from parent p_1 to p_2 between revisions m_1 and m_2. We distinguish seven cases of parent change:

1. *Parent introduction* (PI): p_2 introduced between p_1 and f and p_2 not in m_1;
2. *Parent moved in* (PMI): p_2 introduced between p_1 and f and p_2 exists in m_1;
3. *Parent removal* (PR): p_1 is removed from the feature model (p_1 not in m_2);
4. *Parent move out* (PMO): p_1 is moved (found both in m_1 and m_2);
5. *Parent rename* (PRN): p_1 is renamed: p_1 not in m_2 and p_2 not in m_1;
6. *Feature move* (FM): f is moved from p_1 to p_2 and both exist in m_1 and m_2;
7. *Multiple* (M): a combination of at least two of the above.



(a) Cases of hierarchy per revision (b) Hierarchy refactorings aggregated

Fig. 8. Causes of hierarchy refactoring

Figure 8a shows that there has been significant hierarchy refactoring performed to features during the period, specially in releases 22–23. Curiously, release 22 is considered by the Linux kernel community as a bug-fix release.[4]

Figure 8b reveals that changes of parent are mostly caused by operations on the parent itself, rather than by the explicit moving of a feature. When features are moved, they are moved together with their siblings from a common origin parent to a common destination parent. In fact, for all moves in the period, we found that out of 65 origins, only 4 split its children into more than one destination; and out of 68 destinations, only one came from more than one origin. This suggests that feature model editors, should support moving groups of siblings within a hierarchy (as opposed to only allowing moving subtrees rooted in a single feature to a new place in the hierarchy). Another frequent operation is splicing out features from the hierarchy (to remove them or to move them into another position), without affecting the ancestors and the subtree. To the best of our knowledge neither of operations is directly supported by existing editors.

After further investigation, we found that the underlying reasons for a high level of hierarchy refactoring in releases 22–23 was a consistent replacement of a menu and a config with a menuconfig, consequently eliminating one level of the hierarchy and moving more than 500 features up in the hierarchy. The replacement of menu with menuconfig is made to remove unnecessary mandatory features and replace them with an optional feature capable of enabling/disabling an entire tree hierarchy. This explains the decrease in the average depth over the period, mentioned in Section 4.1.

Adherence to changes in C code. Our sample shows that approximately 15% of edits to the feature model are made together with changes in dependencies in C code. These changes in code are typically code refactoring or bug fixes. The edits to the feature model in 90% of these patches are changes to constraints, following the changes in dependencies in C code.

Build fix. When dependencies in code change and are not immediately reflected in the feature model, developers and users may be unable to successfully compile

[4] http://www.linux-watch.com/news/NS8173766270.html, seen 2010/02-28.

the kernel, and therefore significant development time and user satisfaction is lost. We define a *build fix* to be a delayed adaptation of the model to a change to the source code that appeared in another, earlier patch.

Edits to the feature model in these cases resemble those described in *Adherence to changes in C code*. It is striking that build-fixes are so frequently occurring—clearly indicating need for further research on tools that synchronize the constraints in the model and dependencies in the build system.

Commit logs for changes in constraints indicate that developers do not have enough support for reasoning. Comments range from : "After *carefully examining* the code...", "*As far as I can tell*, selecting ... is redundant" to "we do a select of SPARSEMEM_VMEMMAP ... because ... without SPARSEMEM_VMEMMAP gives us a *hell of broken dependencies that I don't want of fix*" and "it's a *nightmare working out why* CONFIG_PM keeps getting set" (emphasis added). They indicate need for debugging tools for feature models that could demonstrate the impact of edits on the model and on the build system.

Change variability. We have found that there are cases where edits change the configurations with the purpose of adding (or removing) an existing functionality to the feature model, allowing users more configuration options. These operations do not add functional C code; they typically add new configs, make changes to constraints, and add variability to C code by editing `#ifdefs`.

These edits, although few, can be highly complex. Depending on the cross-cutting characteristics of the functionality in question, they may require changes to several different files and locations. For example, commit `9361401` named `'[BLOCK] Make it possible to disable the block layer [try #6]'` required changes to 44 different files and more than 200 constraints.

5 Threats to Validity

External. Our study is based on a single system (Linux). However, we know that this is a mature real world system. As the variability model is an integral part of the Linux kernel, we believe that it should reflect properties of many other long lived models that are successfully evolved, such as operating systems, and control software for embedded systems. We have made initial explorations into the Ecos operating system, which seems to confirm our expectations. Nevertheless, one should not consider our recommendations as representative, since we make them by studying this particular project and not by studying a wide sample of projects.

The Linux development process requires adding features in a way that makes them immediately configurable. As a consequence, it not only enables immediate configurability, but also makes the entire code evolution feature-oriented. Arguably, such a process requires a significant amount of discipline and commitment that may be hard to find in other industrial projects.

Not all projects assume closed and controlled variability model. Many projects are organized in plugin architectures, where variability is managed dynamically using extensions (for example Mozilla Firefox or Eclipse IDE). Our study does not provide any insight into evolution of variability in such projects.

We only look at a fragment of the Linux evolution. We consider this fragment to be relevant since it covers roughly 25% of 20 years long history of Linux. It clearly gives us a glimpse into the evolution of a mature and stable product line.

Internal Validity. Extracting statistical data can introduce errors. We are relying on our own infrastructure for automatic analysis of the Kconfig models. This infrastructure uses the parser extracted from Linux tools, for improved reliability. Also, we are reasonably confident about the quality of the infrastructure, given that we have used it before in another study.

Extracting statistics based on release points may ignore essential information. However we consider any serious fluctuations of our data rather unlikely, should the experiments be carried out at the level of individual patches (partly because our statistics are consistent with each other).

Git allows rewriting histories in order to amend existing commits, for example to add forgotten files and improve comments. Since we study the final version of the history, we might miss some aspects of the evolution that has been rewritten using this capability. However, we believe that this is not a major threat, as the final version is what best reflects the intention of developers. Still, we may be missing some errors and problems appearing in the evolution, if they were corrected using history rewriting. This does not invalidate any of our findings, but may mean that more problems exist in practice.

We use an approximation in interpreting parent change operations above.

Manual classification of edits was feasible and reliable due to excellent comments in Git logs for most of the patches. We increased the robustness of the manual analysis by first running a study on 200 features to identify categories, and then analyzing another set of 200 features selected with uniform probability. An improved study would involve independent cross checking of results.

6 Related Work

The evolution of the Linux kernel between 1994 and 2000 was studied by Godfrey [11], which also found that the Linux architecture is mature and had been growing in a super linear rate, due to growth of driver code. We have found that 3578 patches that modify Kconfig files are driver related. Similarly, Israeli [12] collected several software metrics of the Linux kernel source code from 1994 and 2008, and also observed the functional growth by counting features. Adams studies the evolution of the Linux kernel build system [13] and finds that considerable maintenance to the system is performed to reduce the build complexity, that grows, partly due to the increase in number of features.

Svahnberg and Bosch [14] have studied the evolution of two real software product lines, giving details on the evolution of the architecture and features, closely

related to implementation. They also found that the most common type of changes to the product line is to add, improve or update functionality. Our work, however focuses on the evolution of the model supporting the product line.

Extensive work [2,7,4] addresses issues relevant for detecting edits that break existing configurations and product builds. In [2] an infrastructure is proposed to determine if feature model edits increase, decrease or maintain existing configurations. In [4] a catalog of feature model edits that do not remove existing configurations is presented. Tartler et al. study the Linux kernel [7] and propose an approach to maintain consistency between dependencies in C code and Kconfig.

Work on real case studies on moving to a software product line approach can be found in [15,16,17]. They discuss techniques, processes and tool support needed to make the transition. These works, like [14], focus on product line evolution, not the model, but also suggest that tool support is essential.

7 Conclusion

To the best of our knowledge, this paper is the first to provide empirical evidence of how a large, real world variability model evolves. We have presented the study using the Linux kernel model as our case, collecting quantitative and qualitative data. The following list summarizes the major findings of our work:

- The entire development process is feature driven. In particular the feature model grows together with the code and it is being continuously synchronized with the code. Also the code is systematically retired by eliminating features (and the related implementation). Thus, Linux kernel is a prime example of a mature large scale system managing variability using feature models.
- The model experiences significant growth, in number of features and size. Nevertheless, the dependencies between features only grows linearly with size: the number of features have doubled, but the structural complexity of the model remained roughly the same, indicating a careful software architecture which models features and their dependencies in a sustainable fashion.
- The purpose of most evolution activity is adding new features. The model grows in the process, but only in the width dimension (as opposed to depth). The second largest class of model manipulations are caused by the need to reflect changes in dependencies in source code, in most cases, reactively. Most of the changes at the macro-level are caused by hierarchy refactoring. Constraint refactoring is done for maintenance purposes and is also significant.
- To support evolution, tools should support use cases such as: eliminating features with minimal impact on configuration space, refactoring constraints, propagating dependencies from code to the feature model and tools that allow to manipulate hierarchy easily, while automatically adjusting constraints.

Finally, our investigation proves that maintaining large variability models is feasible and does not necessarily deteriorate the quality of the model. In future work we intend to work on some of the support tools mentioned above.

References

1. She, S., Lotufo, R., Berger, T., Wąsowski, A., Czarnecki, K.: The variability model of the linux kernel. In: VaMoS, Linz, Austria (2010)
2. Thüm, T., Batory, D.S., Kästner, C.: Reasoning about edits to feature models. In: ICSE, pp. 254–264 (2009)
3. Janota, M., Kuzina, V., Wąsowski, A.: Model construction with external constraints: An interactive journey from semantics to syntax. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 431–445. Springer, Heidelberg (2008)
4. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., de Lucena, C.J.P.: Refactoring product lines. In: GPCE, pp. 201–210 (2006)
5. Kästner, C., Apel, S.: Type-checking software product lines - a formal approach, pp. 258–267 (2008)
6. Janota, M., Botterweck, G.: Formal approach to integrating feature and architecture models. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 31–45. Springer, Heidelberg (2008)
7. Tartler, R., Sincero, J., Schröder-Preikschat, W., Lohmann, D.: Dead or alive: finding zombie features in the linux kernel. In: FOSD, pp. 81–86 (2009)
8. Kroah-Hartman, G., Inc., S.L., Corbet, J., LWN.net, McPherson, A.: Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it (2009)
9. Sincero, J., Schröder-Preikschat, W.: The linux kernel configurator as a feature modeling tool. In: ASPL, pp. 257–260 (2008)
10. Bird, C., Rigby, P.C., Barr, E.T., Hamilton, D.J., German, D.M., Devanbu, P.: The promises and perils of mining git. In: Mining Software Repositories (2009)
11. Godfrey, M.W., Tu, Q.: Evolution in open source software: A case study. In: ICSM, pp. 131–142 (2000)
12. Israeli, A., Feitelson, D.G.: The Linux kernel as a case study in software evolution. Journal of Systems and Software, 485–501 (2010)
13. Adams, B., De Schutter, K., Tromp, H., De Meuter, W.: The evolution of the Linux build system. ECEASST (2007)
14. Svahnberg, M., Bosch, J.: Evolution in software product lines: two cases. Journal of Software Maintenance: Research and Practice, 391–422 (1999)
15. Dhungana, D., Neumayer, T., Grunbacher, P., Rabiser, R.: Supporting evolution in Model-Based product line engineering. In: SPLC, pp. 319–328 (2008)
16. Hubaux, A., Heymans, P., Benavides, D.: Variability modeling challenges from the trenches of an open source product line re-engineering project. In: SPLC (2008)
17. Jepsen, H.P., Beuche, D.: Running a software product line - standing still is going backwards. In: SPLC (2009)

Variability Modeling for Distributed Development – A Comparison with Established Practice

Klaus Schmid

Institut für Informatik, Universität Hildesheim
Marienburger Platz 22, D-31141 Hildesheim
schmid@sse.uni-hildesheim.de

Abstract. The variability model is a central artifact in product line engineering. Existing approaches typically treat this as a single centralized artifact which describes the configuration of other artifacts. This approach is very problematic in distributed development as a monolithic variability model requires significant coordination among the involved development teams. This holds in particular if multiple independent organizations are involved.

At this point very little work exists that explicitly supports variability modeling in a distributed setting. In this paper we address the question how existing, real-world, large-scale projects deal with this problem as a source of inspiration on how to deal with this in variability management.

Keywords: Software product lines, variability modeling, eclipse, debian linux, distributed modeling, software ecosystems, global development.

1 Introduction

An increasing amount of software engineering is done in a (globally) distributed way. This has multiple reasons and takes multiple forms. In particular, we see three major forms of distributed development:

1. Distributed by discipline: different parts of a development are done by different (sub-)organizations. These organizational units may be distributed on a world-wide scale.
2. Distributed along a software supply chain: some components are developed by one organization and other components are developed on top of this by a different organization. Such a supply chain may exist either within a single company or across companies. This can in particular be a software ecosystem, i.e., the companies involved are independent on an organizational level and only coupled through the software product (line) [3].
3. Unstructured distributed development: the development distribution structure is not matched to the software structure, i.e., people at different locations work on the same parts of the software.

These forms of software development can be well combined with software product line engineering [14, 5, 12], leading to development organizations of high complexity

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 151–165, 2010.

and the need to synchronize variability across the involved organizations. Distribution type 1 is a rather common organization scheme for large scale development – and thus also for large scale product line development. For example, Nokia develops certain parts of its phone software in labs around the world. Distribution type 2 is actually well known under the name of product populations [22]. Distribution type 3 has also been applied in combination with software product line engineering. It should be noted that while many case studies of product line engineering exist [12, 4] and a significant number of them even deals with a distributed stetting, very little work explicitly addresses the issue of distributed variability modeling.

In this paper, we focus on the question how distributed development impacts variability management, respectively what characteristics make a variability management approach particularly suited for distributed development. As only distribution types 1 and 2 relate development structure and distribution, we will focus on those approaches to distributed development. We will also address in particular the situation of software ecosystems, i.e., development is distributed across multiple organizations which are only loosely coupled [3].

The remainder of this paper is structured as follows: in Section 1.1 we will introduce the key research questions of this paper and in Section 1.2 we will discuss related work. Section 2 will then introduce the case studies that we chose to analyze, Debian Linux and Eclipse. In Section 3 we will discuss to what extent our case studies are reasonable cases of variability management. On this basis we will discuss in Section 4 the main concepts that can be taken from these case studies to support distribution and in particular distributed variability management. Finally, in Section 5 we will summarize and conclude.

1.1 Research Questions

The main goal of this paper is to improve the understanding of how variability management can be effectively supported in the context of distributed development. As a basis for answering this question, we decided to analyze some existing highly configurable infrastructures in order to identify what works in practice. The examples we draw upon are Debian Linux [1] and Eclipse [9], respectively their package management. Both are large-scale, well-known projects, which do heavily rely on highly distributed development. Moreover, many independent organizations can contribute packages, realizing a rather low level of interdependence among the organizations. Thus, they show that these approaches can in particular be applied in the context of a software supply chain (respectively a software ecosystem).

Of course, it is non-trivial that the configuration approach which is used in the package management systems of Eclipse and Debian Linux can be compared to variability management approaches at all. In order to address this concern we will make an explicit comparison of these approaches with variability management techniques like feature modeling [11].

In summary, we will address the following research questions in this paper:

(RQ1) Can existing package management approaches like those of Debian Linux and Eclipse be regarded as a form of variability management?

(RQ2) Which concepts in these package management approaches can prove useful to handle distributed development, if any?

The main focus of our work will be on the aspect of distribution. Thus, while we will address the aspect of expressiveness, it is our main goal to identify concepts that might help variability management approaches to better cope with distribution.

1.2 Related Work

In this paper we analyze two existing software configuration systems and compare them with variability management approaches. So far such analysis of existing software configuration systems have been performed surprisingly few. A notable exception is the analysis of the Kconfig system used in Linux kernel development by She et al. [20].

A difference between the case studies we use here and case studies like [20] is that here configuration is performed rather late, i.e., at installation time. However, this is in accordance with recent developments in the product line community where later binding of variability is increasingly common (e.g., [23]). This is sometimes termed dynamic software product lines [10] and seems to be very relevant to software ecosystems [3].

A major aspect of distributed development is that the variability management needs to be decomposed as well. This has so far received very little attention. A notable exception is the work by Dhungana et al. [8]. They introduce the notion of decision model fragments which can be manipulated independently and integrated at a later point. They explicitly mention distributed development (type 1) as a motivation for their work. The feature diagram references mentioned by Czarnecki et al. [6] are also related as it seems their approach can also be used to support distributed development, though this is not made explicit.

Recently, Rosenmüller et al. also addressed the issue of integrating multiple product lines into a single, virtual variability model under the name of multiple product lines [16, 15]. A major disadvantage of their approach from the perspective of our discussion here is that it requires a single central point of control (the integration model). We expect this to problematic, in particular in the context of open variability, e.g., in software ecosystems [3].

In this paper, we do not aim at introducing a new approach. Rather we focus on how existing approaches from industrial practice deal with the problem of distributed development of a software product line, respectively a software ecosystem [3]. We will analyze these approaches, show that they are comparable to existing variability management approaches, while offering at the same time new insights that should be taken into account for designing future distributed variability management approaches.

2 Analysis of Case Studies

In this section, we provide an overview of the two case studies we will use as a basis for our analysis: the Debian Package Manager [1] and the Eclipse Package Manager [9]. While these are certainly not the only relevant, distributed software configuration systems, we restricted our analysis to these two as they are well-known, widely used and documentation of them is easily available. Also they are clearly examples of the problem of distributed and rather independent development. In particular they are used as a means to realize a software ecosystem [3]. We also considered to include Kconfig in our analysis, but decided against this for multiple reasons: first of all a

good analysis like [20] already exists, although the focus was in this analysis not on distribution aspects. Second Kconfig supports the configuration of the Linux Kernel, which is released in a centralized way. Thus, we expected to learn less from this approach compared to the ones we selected for more general distribution models like software supply chains (cf. distribution 2) or software ecosystems.

Both the set of all Linux installations, as well as the set of possible Eclipse configurations can be regarded as a rather open form of product line respectively a software ecosystem. It is a product line, as the customer has even with a standard distribution of Linux such a large range of possible configurations, that actually each installed system can have a unique configuration. Further, it can be regarded as an open product line, as additional systems can refer to and extend the existing base distribution enlarging the overall capabilities and configuration space. It is this form of openness of the variability model which is particularly interesting to us from the perspective of distributed development.

It is important to note – and we will see this in more detail below – that the approach used by these management systems is completely declarative. This makes them rather easy to compare to variability management and is different from approaches like makefiles, which contain procedural elements and can thus not be directly compared to declarative variability management approaches.

Some concepts are common to both the Eclipse and the Debian Linux package management approach, thus, we will first discuss their commonalities, before we discuss the specifics of both systems in individual subsections.

Both Linux and Eclipse are actually aggregates that consist of a number of packages. So-called distributions are formed, which are collections of packages which are guaranteed to work well together. In the case of Linux the user can select a subset of these packages for installation, while in the case of Eclipse a specific distribution forms a set which is installed as is. The packages in a distribution are selected (and if necessary adapted) to work well together. This is the main task of the organization that maintains a distribution. It should be noted, however, that this organization is not necessarily responsible for the actual development. It is thus more appropriate to compare it to an integration department in traditional software development. There exists a configuration model as a basis for defining a specific installation. This is included in a distribution and is stored in a distributed (per-package) fashion, as we will discuss later.

As the various packages that belong to a distribution can be developed by a large number of different development organizations, we have already here the situation of distributed development, although there is an explicit harmonization step in the form of the maintenance of the distribution. It should be noted, however, that this is harmonization is not needed per se, as both approaches are open to the inclusion of arbitrary third party packages. The step of distribution formation mainly plays the role of quality assurance.

In both cases a package manager exists, which uses the descriptive information from the packages to provide the user[1] with configuration capabilities. Users can also install additional packages (e.g., from the internet) which are not part of the initial

[1] We will use the term *user* to denote any individual who performs a configuration process. In practice the user will typically fill the role of an administrator.

distribution. As these packages can be developed completely independently, this further enforces the notion of distributed development.

The user can use the package manager to create the final configuration. This step results in a specialized installation. It is peculiar of this approach that the binding time is rather late (i.e., the individual packages already contain compiled code), however, this is not a must, as even source code can be part of packages. This is in particular the case in the Linux environment. This difference in binding time may seem unusual, as typically product line engineering is equated with development time binding. However, already in the past different approaches have been described that go beyond this restriction (e.g., [23, 21, 17]). This is actually most pronounced in the context of dynamic software product lines (DSPL) [10].

2.1 Debian Linux Package Management

The installation packages for the Debian package manager consist of the following parts [1]:

- *Control-File:* the control file contains the necessary dependency information as we will discuss below.
- *Data File:* the data file contains the actual data to be installed. This can be source code, binary files, etc.
- *Maintenance scripts:* these are used to perform maintenance actions before and after the reconfiguration takes place.

The key information we are interested in here is contained in the *control file* as this contains all relevant dependency and configuration information. The control file provides administrative information like the name and the version of the package, but also dependency information. It also defines the required disk space and the supported hardware architecture. This constrains the situations when it can be installed.

For the dependency information in a package seven different keywords are defined: *depends, recommends, suggests, enhances, pre-depends, conflicts,* and *replaces*. The semantics of these keywords overlaps as discussed below:

- *Depends:* this keyword expresses that the following text provides information on other packages that are required for the proper use of this package. This can be combined with a version qualifier with the implication that the package requires this version (or a later one) of the package.
- *Pre-Depends:* similar to depends it defines that another package is needed *and* must be fully installed prior to installation of the current one. Thus in addition to the dependency information it provides execution information for the package manager.
- *Recommends:* this expresses that the package is typically installed together with the recommended packages. However, this is not a strict dependency, but rather a hint for the configuration, that the recommended packages should as well be installed (but a configuration will also be successful without them).
- *Suggests:* this expresses that the suggested packages could be installed together with the current one. This should be interpreted as a hint, it is similar to recommends, but should be interpreted in a much weaker form.

- *Enhances:* this defines that the package can be used together with the enhanced packages. This should be interpreted as the inverse relationship to *suggests*.
- *Conflicts:* this expresses that the current package cannot be used in combination with the mentioned packages.
- *Replaces:* this expresses that installation of files in the package will actually replace or overwrite information from the referenced packages.

As we can see the different keywords actually combine different aspects in a non-systematic may. These aspects are:

- *Dependency and conflict information:* this can be compared to the information typically contained in a variability model.
- *User guidance:* some information is only meant as a hint to the user of the configuration system (e.g., suggests, enhances). This is actually ignored by some package management tools [1].
- *Execution information:* information like pre-depends actually influences how the package manager performs the actual installation process.

Our main interest is of course on the dependency and conflict information. All of the relationships (except for *replaces*) that are introduced by the control files have some aspect of dependency and conflict information, although in some cases (e.g., suggests) this is not strict in the sense that the given guidance can be ignored by the user, respectively the installation system.

The remaining two parts of a package are the data file and the maintenance scripts. The *data file* is basically a packed archive. There are actually two slightly different formats, depending on whether the package contains source code or binaries (e.g., executables). Unpacking the archive generates all the necessary files, including directories, etc. This implies an all or nothing semantics, i.e., either the whole data contained is added to the installation or the package is not installed.

Finally, there are *maintenance scripts.* These are mainly important as the package manager may run while the system is actually running. The scripts are then used to start and stop services to allow their (re-)installation. Another application of these scripts is to customize the configuration process. For our analysis these scripts are not of further interest.

2.2 Eclipse Package Management

The Eclipse package management provides two concepts that are relevant to our analysis here: *feature* and *plug-in* [9]. The feature in the Eclipse terminology is a coarse-grained capability. Actually the typical user installation consists of only three major features [9]: platform, java development tooling, plug-in development tooling.

A feature by itself does not contain any functionality, rather it describes a set of plug-ins that provide the functionality. In addition to acting as a sort of container for the actual plug-ins it provides management information like where to get updates for the feature.

A plug-in consists of a so-called *manifest* which provides a declarative description of the plug-in and the relevant resources. The resources contain a set of java classes, which implement the functionality, but may contain also other resources like scripts, templates, or documentation.

The main part, we are interested in here, is the manifest. It declares plugin name, id, version, provider, etc. It also defines the dependency information for the plug-in. Dependencies are defined in the *requires*-section of the plug-in manifest [9]. This section declares other plug-ins that are needed for successful usage of the current one. The plug-ins can be further annotated with version information.[2] In addition the *requires*-information can be further refined as *optional*. The semantics of an optional requires is that the referenced plug-in should be integrated into the platform if it is found, but if it is not found the installation of the current plug-in is still possible (as opposed to a pure requires). On the other hand, Eclipse does not provide any way to express that a plug-in is mutually exclusive (conflicts) with another plug-in. The requires-information can be further refined by making restrictions with respect to the versioning information. This is supported by a proposal for the semantics of the version numbering by Eclipse. Specific relations on the versions include: perfect match, equivalent, compatible, greaterOrEqual.

The Eclipse feature and plug-in mechanisms also support some sort of modularization. This is expressed by exports and extension points.

The classes that make up the plug-in can also be exported, enabling other plug-ins to explicitly refer to them.

In addition the manifest may declare extension points and extensions. The extension point architecture of Eclipse is a core part of its extensibility. Any plug-in may define extension-points. This means it will allow explicit, external customization of its functionality. A plug-in may also refer to an extension point and extend it. Typical examples for extension points within the Eclipse-IDE are menu entries or additional views.

3 Package Managers as a Form of Variability Management

In this section, we will focus on the question of whether the package managers, described above can be seen as a form of variability management (RQ1). In order to characterize variability management several formalizations of variability modeling, like [19, 7, 2] have been developed. As we will see, the package managers only support rather simple concepts of variability management, thus a simplified treatment of variability management is sufficient in this context.

3.1 Variability Management Concepts

As a first step, we need to establish a mapping between the typical concepts used in package managers and in variability management.

If we take as a basis for variability management the concepts from feature modeling, as they are described in a formal definition like [19], we find that a feature model (represented by a feature diagram) can be characterized in the following way (we use here the terminology introduced in [19]):

- *Graph Type:* this can either be a graph or a tree.
- *Node Type:* possible node types; these are Boolean functions like *and*, *or*, *xor*. This describes how nodes can be decomposed. Also cardinalities *(card)* are classified as node types.

[2] For identifying required versions it is possible to define constraints as exact matches, compatible, etc. This is implemented using a specific version naming scheme.

- *Graphical Constraint Type:* is a binary Boolean operator, e.g., *requires* (⇒) or *mutex* (|).
- *Textual Constraint Language:* is a subset of the language of Boolean formula and is used to describe the allowed compositions.

In addition, it should be noted that edges within a feature diagram denote a decomposition. The allowed forms of decomposition are expressed by the node type.

The mapping of the main concepts in package managers to such a variability modeling language are not straight-forward as the relationships are somewhat different and the package managers do not support a diagram notation. Thus, we will discuss this mapping here explicitly. In support of the discussion Figure 1 illustrates the main concepts we will deal with.

The first major difference between the package management approaches and feature diagrams is that the approaches are textual not graphical. We can thus ignore the difference between the graphical constraint type and the textual constraint language. The nodes in the package management approaches correspond to packages (in Debian Linux), respectively plug-ins in Eclipse. It should be noted that the concept of features as it is introduced in Eclipse is rather coarse-grained and describes actually a conglomerate of plug-ins while the basic level on which dependencies are expressed are on the level of individual plug-ins. This is shown in Figure 1 by illustrating packages, but having the relations on the level of the individual contained units.[3]

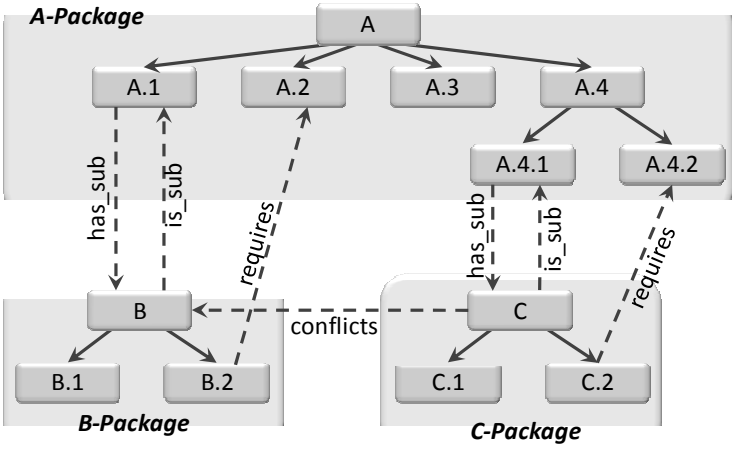


Fig. 1. Main concepts of variability in package managers

A major concept in most feature modeling approaches is that the features are decomposed in the form of a graph or tree. This is shown in Figure 1 as *has_sub* and is described by the node-type as defined above. It should be noted, however, that such a decomposition approach is not part of all forms of variability management approaches.

[3] This corresponds closely to the situation in Eclipse, in the Debian Linux situation the packages shown in Figure 1 have no correspondence.

For example, some forms of decision modeling [18] or the approach described in [14] do not rely on decomposition. Both package managers do also not have the concept of decomposition. However, still often a tree-like or graph-like dependency structure is introduced. This can be done using the *requires*-links. This can be seen as *has_sub*-relation shown in Figure 1. On the other hand, as the various packages can be selected individually and are interrelated mainly by requires-relations, this relation can also be regarded as the inversion of the *has_sub*-relation, i.e., the *is_sub*-relation. Thus, in both approaches, both relations *has_sub* and *is_sub* are replaced by requires, if they are represented at all.

3.2 Analyzing Package Management as Variability Management

We will summarize the expressiveness of the two package management approaches and will use this as a basis to compare them with variability management as described by the characterization from [19].

As discussed in Section 2.1 the Debian Linux package management has as main concept the package. We can equate this with a node. As a basis for dependency management the relations *depends* and *conflicts* can be regarded. The *depends*-relation can be equated with requires. The *conflicts*-relation is not exactly the same as the *mutex*-relation, defined in [19], as it is not symmetrical, thus we keep the name *conflicts* in Figure 1. However, *conflicts* effectively emulates the *mutex*-relation as it does not allow a configuration in which both packages participate. Noteworthy is also the *recommends*-relation as it has the same direction as the *has_sub*-relation. However, it is weak in the sense that it does not require the installation. The other relations defined by the Debian Linux package management approach (*pre-depends, suggests, enhances, replaces*) are variations of the mentioned relations, but augment it with additional user advice or execution information. They do not provide any new information from a logical point of view.

Below, we discuss the comparison in more detail:

- The most striking difference is probably that the decomposition hierarchy which is typical for feature diagrams is not directly a part of the dependency management defined by the package management approaches. As a consequence the resulting structure is not one continuous graph, but is rather a set of graphs. Further, the hierarchy in the sense of decomposition cannot be directly expressed, but can only be simulated using a requires-relationship. While both aspects seem unusual for feature-based variability management, they exist in other variability management approaches like [18, 14, 7] as well.
- The *and*-, *or*-node types can be expressed as described in Table 1. The *xor*-node type can only be represented for the Debian Linux package manager by means of the conflict relationship. This is not available for Eclipse and can thus not be simulated there. More advanced concepts like cardinality do not exist in either approach.
- Constraints are always textual constraints, as there is no graphical notation for both approaches, thus we discuss Graphic Constraint Type and Textual Constraint Type

together.[4] Again the Debian Linux approach allows the representation of *mutex*- and *requires*-constraints (using *conflicts* and *depends* relations, respectively). The Eclipse package manager falls short as it can represent the *requires*-relation, but not the *mutex*-relation. More complex combinations (in the sense of complex formula) are not available in either approach.

Table 1. Comparison of package management and a characterization of feature models ([19])

Concept [19]	Debian Linux Package Manager	Eclipse Package Manager
Graph Type	Graph*	Graph*
Node Type	and, or – all elements that are required must be installed, but weaker versions like suggests actually provide optionality xor – can be simulated by using the conflicts relation	and, or – the requires relationship can be augmented with an optional modifier
Graphical Constraint Type (GCT)	requires – is supported using depends mutex – the conflicts relationship is a directed variant of mutex	requires – exists as a relation mutex – does not exist nor can be simulated
Textual Constraint Language	From a logical point of view only the concepts mentioned under GCT are supported, although the language is completely textual. There exist extensions like references to specific version, which do not exist in variability modeling techniques for product lines.	

*: As there need not be connections between nodes that are installed, either induced by requires or any other relations, it might actually be more appropriately regarded as a set of graphs.

If we accept the use of the requires-relation to describe the decomposition, we can deduce from [19] that the expressiveness of the Debian Linux approach is at least similar to FODA [11].[5] More problematic is the Eclipse package manager, which does not support a form of exclusion (similar to alternatives, mutex-relations, conflicts-relation, etc.). As a consequence, we need to accept that this approach is truly weaker than other variability management approaches.

According to the above comparison we can deduce several findings. The first and most important is that we can answer (RQ1) with yes for the Debian Linux approach,

[4] Again, it should be noted that this also exists in other variability modeling approaches. For example the approach described in [18] only provides a semantic approach without prescribing a specific representation (graphical or otherwise).

[5] This is not fully correct, as FODA allows parameterization, which is not present in the Debian Linux model. However, this is also not part of the analysis given in [19].

albeit it truly provides only minimal expressiveness (comparable to FODA). For Eclipse, answering (RQ1) is not easy, as the expressiveness is less powerful than any of the variability management approaches given in the literature, due to the lack of the *mutex*-relation. We will thus answer (RQ1) for Eclipse only as partially, however, many elements of a variability management approach exist. Thus, it forms a valid basis for our comparison in this paper.

In summary, we can say that both approaches lack in comparison with modern variability management approaches significant expressiveness. Examples for this are cardinalities, complex constraint conditions, etc. From this perspective it is surprising that both approaches work very successful in practice. One reason for this is certainly their limited application domain.

4 Concepts in Package Management That Support Distribution

After discussing whether the two package managers can actually be regarded as a form of variability management, we will now turn to the question what concepts exist in the two package managers which may prove useful for variability management, in particular in a distributed setting (RQ2).

We will structure our discussion along the following topics:

- Decomposition
- Version-based dependency
- Information Hiding
- Variability Interfaces

Decomposition: Probably the most immediate observation one can make when analyzing the package managers is that they take a decomposition of what shall be managed for granted. Of course this flows well with distribution. In Debian Linux the basic concept is a package, but it should be noted that the contribution of a distributed part of the development is often contained in several packages that are related with each other (e.g., an implementation and a source package). In Eclipse the feature concepts defines such a unit of distribution and may contain several plug-ins. However, also in the Eclipse case sometimes several features are developed and distributed together, as the feature is also a configuration unit.

From the perspective of distributed variability management this decomposition also leads to a decomposition of the variability model as this enables to assign responsibility for different of the variability model to different teams. We believe this is one major characteristic to support distributed variability management. Further, if we look at the way the relations are typically used within the package management approaches, we see that the *has_sub*-relation is typically not used across packages that build on each other, but rather *requires* is used in the form of the *is_sub*-relation in Figure 1 (or across the hierarchy as also shown in Figure 1). This leads to the package that builds on top of another one to know the package on which it is building, but not vice versa. Thus a platform can be built without the need to know everything which will build on top of it at a later time. We regard this as an important difference to the decomposition hierarchy in feature models and term it *inversion of dependency*. It should be noted, however, that for other variability management approaches that do

not have a decomposition hierarchy this is straightforward. We regard this *inversion of dependency* as very important for developing future software ecosystems, using product line technologies [3].

Version-based dependency: the capability that all packages may have versions and that the dependency management may use the version information to define what combinations are acceptable is very useful to reduce the coupling between packages. Thus, packages may still work together, even if their content changes and this may be explicitly expressed in the relations. This is particularly prominent with the Eclipse package manager, which even defines different forms of changes (and can put constraints on the acceptance of the different compatibility levels). This enables the specification of the degree of decoupling of the development of the different packages that is possible.

Information Hiding: Of course explicit variability management always leads to some form of information hiding as the potential configurations only depend on the features defined in the variability model, not on the base artifacts. However, Eclipse goes one step further by explicitly defining what parts of the packages will be visible to other packages. This concept does not exist in classical variability modeling, as there no information hiding is induced on the basic artifacts. It is interesting to note that this is similar to packages in the UML which also allow to restrict the visibility of their content. Also information hiding in Eclipse goes hand in hand with explicit variability interfaces.

Variability Interfaces: A very interesting mechanism within the Eclipse package management approach is the extension point approach. Plug-ins can announce specific extension points where further variability is possible and can be resolved by other packages. Typical examples of this are menus, views, etc. that can be augmented by further plug-ins. Eclipse even introduces a schema definition for extension points. While this would allow the definition and verification of the parameterization of the extension point, this schema is currently not yet used in the Eclipse implementation according to the latest description [9].

The concepts of *decomposition*, *information hiding* and *interfaces* are all well-known. They are typical of what we call today *modularization* [13]. However, if we compare this with almost all existing variability modeling approaches, we have to recognize that they are monolithic in nature (for a discussion of the exceptions see Section 1.2). It seems reasonable to assume that the modularization concepts that are useful in the construction of every software systems, in particular in a distributed manner are as well useful in distributed variability modeling. This is emphasized by the success that the discussed package management systems already have in industrial practice today were they provide a foundation for respective software ecosystems.

The concept of *version-based dependency* further supports the decoupling of the variability packages. Both discussed approaches posses this capability in some form. Finally, the *inversion of dependency*, i.e., that only the refining variability package needs to know the refined package, but not vice versa seems rather useful, as it further decouples distributed development.

Thus, we can answer (RQ2) positively: there are certain concepts that have been introduced in package management approaches that are helpful for distributed vari-

ability management in software product line engineering. The key concepts we could identify are: decomposition, version-based dependency, information hiding, variability interfaces, and inversion of dependency. Out of these only decomposition has to our knowledge been applied so far [15, 8, 6]. The other concepts introduce some form of modularization to provide a better basis for decoupling in distributed product line development.

5 Conclusions

In this paper, we analyzed existing, large-scale, real-world package management approaches to identify concepts they can offer for distributed variability management. We established (RQ1) that these approaches can indeed be interpreted as a form of variability management as they support the declarative definition of variation constraints. The concepts they support for dependency management can be mapped onto existing variability management approaches. However, we had to recognize that only very fundamental concepts are realized. Thus, these tools may significantly profit by integrating more advanced capabilities from modern variability management approaches.

On the other hand these package management approaches have been developed from the beginning to support distributed development and integration. Both provide today the basis of a software ecosystem of their own. As a result they offer a slightly different approach and concepts that can be well integrated into product line engineering to support improved modularity of variability management. We can thus answer our second research question (RQ2) positively. The concepts we identified are:

- Decomposition
- Version-based dependency
- Information hiding
- Variability interfaces
- Inversion of dependency

However, it should be recognized that these concepts, while certainly useful for augmenting existing variability management techniques, they can still be further improved. For example, it would be useful to provide a formal definition of variability interfaces. As a result of the introduction of the above concepts together with further work, we expect that in the future we will arrive at a meaningful theory on the modularization of variability models [13]. We expect that such a theory will be particularly relevant in the context of open and distributed variability as is required for software ecosystems [3].

As a next step we plan to extend our work and will address in particular software ecosystems more in depth. We will also extend our basis of analysis further to cover a larger range of existing approaches. The results of this analysis will then drive the development of an integrated approach for dependency management that combines the strengths of established practical approaches with the best of existing, sophisticated variability management techniques.

References

[1] Aoki, O.: Debian Reference, (2007), http://qref.sourceforge.net/Debian/reference/reference.en.pdf (last verified: 13.3.2009)

[2] Benavides, D.: On the automated analysis of software product lines using feature models. A framework for developing automated tool support. PhD thesis, University of Seville, Spain (2007)

[3] Bosch, J.: From software product lines to software ecosystems. In: Proceedings of the 13th Software Product Line Conference, pp. 111–119 (2009)

[4] Catalog of software product lines, http://www.sei.cmu.edu/productlines/casestudies/catalog (last verified: 13.03.2010)

[5] Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston (2002)

[6] Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multi-level configuration of feature models. Software Process Improvement and Practice 10(2), 143–169 (2005); Special Issue on Software Product Lines

[7] Dhungana, D., Heymans, P., Rabiser, R.: A formal semantics for decision-oriented variability modeling with dopler. In: Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2010), pp. 29–35 (2010)

[8] Dhungana, D., Neumayer, T., Grünbacher, P., Rabiser, R.: Supporting the evolution of product line architectures with variability model fragments. In: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture, pp. 327–330 (2008)

[9] The Eclipse Foundation. Eclipse 3.1 Documentation: Platform Plug-in Developer Guide (2005), http://www.eclipse.org/documentation (checked: 13.3.2009)

[10] Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. Computer 41(4), 93–95 (2008)

[11] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21 ESD-90-TR-222, Software Engineering Institute Carnegie Mellon University (1990)

[12] van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering. Springer, Heidelberg (2007)

[13] Parnas, D.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)

[14] Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)

[15] Rosenmüller, M., Siegmund, N.: Automating the configuration of multi software product lines. In: Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2010), pp. 123–130 (2010)

[16] Rosenmüller, M., Siegmund, N., Kästner, C., ur Rahman, S.S.: Modeling dependent software product lines. In: GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE), number MIP-0802, pp. 13–18. University of Passau (2008)

[17] Schmid, K., Eichelberger, H.: Model-based implementation of meta-variability constructs: A case study using aspects. In: Proceedings of VAMOS 2008, pp. 63–71 (2008)

[18] Schmid, K., John, I.: A customizable approach to full-life cycle variability management. Science of Computer Programming 53(3), 259–284 (2004)

[19] Schobbens, P.-Y., Heymans, P., Trigaux, J.-C.: Feature diagrams: A survey and a formal semantics. In: Proceedings of the 14th IEEE Requirements Engineering Conference (RE 2006), pp. 139–148 (2006)
[20] She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: The variability model of the linux kernel. In: Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2010), pp. 45–51 (2010)
[21] van der Hoek, A.: Design-time product line architectures for any-time variability. Science of Computer Programming 53(30), 285–304 (2004); Special issue on Software Variability Management
[22] van Ommering, R.: Software reuse in product populations. IEEE Transactions on Software Engineering 31(7), 537–550 (2005)
[23] White, J., Schmidt, D., Wuchner, E., Nechypurenko, A.: Optimizing and automating product-line variant selection for mobile devices. In: Proceedings of the 11th Annual Software Product Line Conference (SPLC), pp. 129–140 (2007)

Variability Management in Software Product Lines: An Investigation of Contemporary Industrial Challenges

Lianping Chen[1] and Muhammad Ali Babar[2]

[1] Lero - the Irish Software Engineering Research Centre, Limerick, Ireland
lianping.chen@lero.ie
[2] IT University of Copenhagen, Denmark
malibaba@itu.dk

Abstract. Variability management is critical for achieving the large scale reuse promised by the software product line paradigm. It has been studied for almost 20 years. We assert that it is important to explore how well the body of knowledge of variability management solves the challenges faced by industrial practitioners, and what are the remaining and (or) emerging challenges. To gain such understanding of the challenges of variability management faced by practitioners, we have conducted an empirical study using focus group as data collection method. The results of the study highlight several technical challenges that are often faced by practitioners in their daily practices. Different from previous studies, the results also reveal and shed light on several non-technical challenges that were almost neglected by existing research.

1 Introduction

Software intensive systems in a certain domain may share a large amount of commonalities. Instead of developing each product individually, software product line engineering looks at these systems as a whole and develop them by maximizing the scale of reuse of platforms and mass customization [20]. Thus, it is claimed that Software Product Line (SPL) can help reduce both development cost and time to market [15]. A key distinction of Software Product Line Engineering (SPLE) from other reuse-based approaches is that the various assets of the product line infrastructure contain variability, which refers to the ability of an artifact to be configured, customized, extended, or changed for use in a specific context [3]. Variability in a product line must be defined, represented, exploited, implemented, and evolved throughout the lifecycle of SPLE, which is called Variability Management (VM) [15]. It is a fundamental undertaking of the SPLE approach [15].

VM in SPL has been studied for almost 20 years since the early 1990s. Feature-Oriented Domain Analysis (FODA) method [10] and the Synthesis approach [11] are two of the first contributions to VM research and practice. Since then diverse methods/approaches have been proposed [7]. The goal of all these research efforts should be to help practitioners to solve their real problems. Hence, there is a vital need to explore how well the VM body of knowledge solves the problems faced by industrial practitioners, and what are the remaining and (or) emerging issues. Such an effort can help to update the understanding of issues and VM challenges in SPL practice. Such

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 166–180, 2010.

an understanding is expected to help researchers to direct their research efforts towards real and high priority issues and challenges in the industry, and thus can provide practitioners with more support for VM and improve their productivity. As such, the specific research question that motivated this study was:

- What are the contemporary industrial challenges in variability management in software product lines?

The goal of this paper is to report the results of an empirical study aimed at identifying issues and challenges of VM in SPLE faced by industry practitioners. This paper is organized as follows. Section 2 provides the details of the research methodology used for this research. Section 3 presents the findings of this study. Section 4 discusses the findings from analysis of the data gathered during the focus group discussions with respect to the published literature on variability management in software product lines. Section 5 mentions some of the potential limitations of the reported study and its findings and Section 6 finishes the paper with a brief discussion about the outcomes from the reported study and future work in this area.

2 Research Method

We conducted an empirical study using focus group research method in order to identify the issues and challenges of VM in SPLE faced by industry practitioners in their daily activities. We decided to use the focus group research method because it is a proven and tested technique to obtain the perception of a group of selected people on a defined area of interest [1, 13, 24]. In the following sub sections, we describe the process of this study according to the five steps involved in the focus group research method.

2.1 Define the Problem

In this step, we defined the research problem that needed to be studied by using the focus group research. The research problem was derived from our research goal (i.e., to gain an understanding of issues and challenges of VM in SPL in practice) as described in Section 1.

2.2 Plan Focus Group

In this step, we set the criteria for selecting participants, decided the session length, designed the sequence of questions to ask during the session[1], and prepared documents to provide the participants with the study background, objectives, and protocols.

2.3 Select Participants

In this step, we selected participants according to the criteria devised during the planning stage. In order to gain insights into the VM challenges in practice, we followed the following criteria for selecting the participants:

[1] One of our colleagues also participated in designing the sequence of questions to ask during the session.

- experience of variability management in practice,
- knowledge and expertise of issues/challenges of variability management, and
- willingness to share their experiences and candid opinion.

According to the selection criteria, our study needed practitioners with industrial experience in VM in SPLE. Such practitioners are usually very busy and are not likely to respond to invitations from unfamiliar sources. Thus, a random sampling was not viable.

Consequently, we decided to use availability sampling, which seeks responses from those who meet the selection criteria and are available and willing to participate in a study. The International Software Product Line Conference (SPLC) attracts a large number of practitioners every year. We sent invitation emails to practitioners who were going to attend SPLC 2008 and met our selection criteria.

2.4 Conducting the Focus Group Session

We held three focus group sessions, each of them lasting approximately one hour. The flow of the discussion was designed to be as natural and as easy to follow as possible. Each session started with a brief introduction of the participants and researchers. Then the discussion flowed through a predefined sequence of specific topics related to the challenges in different phases of SPLE, i.e., requirements phase, architecture phase, implementation phase, testing phase, and any other aspect of VM in SPLE. The separation between phases is based on the SPLE framework presented by Pohl et al. [20]; however, to not complicate the discussion flow, we decided not to separate the discussions on domain engineering and application engineering in each phase. The sessions were audio recorded with the participants' consent.

2.5 Data Analysis

In this step, we transcribed the recorded discussion and coded the transcription. The focus group sessions of this study resulted in approximately three hours of audio recording. The audio recording was transcribed by transcribers. In order to verify that there was no bias introduced during the transcription, the first researcher randomly checked several parts of the transcription. No significant differences were found.

To analyze the transcribed data, we performed content analysis and frequency analysis. We followed the iterative content analysis technique, which is a technique for making replicative and valid inferences from data to their context [14], to prepare qualitative data for further analysis. During content analysis, we mainly used Strauss and Corbin's [26] open coding method. With this method, we broke the data into discrete parts, and closely examined and compared them for similarities and differences. Data parts that are found to be conceptually similar in nature or related in meaning were grouped under more abstract categories. The coding was performed by the first researcher and checked by the second researcher.

To identify the relative importance of the challenges' influence on industrial practices in variability management, we performed frequency analysis on the transcribed data for the high level themes.

3 Results

In this section, we present the results of the study. We first present the demographics of the participants, then present the issues reported by the participants, and finally describe the frequency analysis.

Table 1. Demographic information about the participants

ID	Title	Experience	Country	Domain	Company size	Type of company
1	Principle member of research staff	8+ years in SPL; has been working with 40 SPLs.	Finland	Mobile phones	112,262	In-house
2	Senior member of the technical staff; Principal	Worked in SPL since 1990; consulted various companies.	USA	Various	<50	Consultant
3	Project manager in SPL	5 years in SPL	Spain	Embedded	51-200	Consultant
4	Software engineer, SPL supporter	SPL initiative started about 6 or 8 months	USA	Defence, aerospace	106,000	In-house
5	Chief software architect.	20 years in SE; 7 years in SPL	USA	Embedded	73,000	In-house
6	Software architect and software development process manager	Introduced the SPL approach 4 months ago;	Germany	Embedded	4,000	In-house
7	Director	10 years in SPL; consulted various banks and insurance companies	Australia	Finance	40	Consultant
8	Research scientist	Work three days per week in the company since 2004	Netherlands	Healthcare	123,801	In-house
9	Software architect	25+ years in SD; around 5 years in SA and SPL.	USA	Embedded	263,000	In-house
10	Global software process and quality manager	6 years in SPL	Switzerland	Embedded	128,000	In-house
11	Senior scientist	About 8 years in SPL	Netherlands	Embedded	33,500	In-house

3.1 Demographics and Frequency of Participants' Participation

Table 1 shows the profile of the participants of the focus group sessions. There were 11 participants in the focus group sessions. The majority of them were holding senior positions in their respective organizations and were playing important roles (e.g. responsible for, advocator, and introducer) in their organizations' SPL adoption and management practices. The participants also had good knowledge of their companies'

SPL initiatives. It is worth to note that some of them had the title of "researcher"; however, they were working in research centers in industrial companies rather than academic research institutes. They usually had good knowledge of the VM challenges in their organizations. So we considered them as SPL practitioners in this study.

Each of the participants came from a different company. These 11 companies varied in type, size, domain, and geographical area. While the majority of the companies were in-house development units, there were also three consultancy companies. All the in-house development companies were of large size in terms of the number of employees[2]. The consultant companies were of a small to medium size; however, the participants from these three consultancy companies had worked with various other large companies, so they brought in their experience with those various companies as well. The majority of the participants' companies were working in embedded systems; however, there were also representatives from other domains like finance and telecommunications. The companies where the participants came from are located in seven different countries covering three continents (i.e., America, Europe, and Australia).

The demographics information about the participants of our study gives us confidence that we gathered data from practitioners who were knowledgeable about VM challenges based on their experience in practice. Furthermore, although the number of the participants is not high, they came from 11 different companies (most of them had extensive experience in VM) and 7 different countries. So their views can be considered representative of the VM challenges faced by broad practitioners in industry with similar characteristics.

Table 2 summarizes the amount of participations by each participant. The number in the cell indicates the number of speeches from a particular participant. It can be observed that there were no dominant speakers during the discussion. Every participant got almost equal opportunity to share his/her experience and opinion on different aspect of VM in SPL.

Table 2. Frequencies of speeches by each participant

Participant ID	1	2	3	4	5	6	7	8	9	10	11
Frequency of speeches	43	41	19	37	37	29	25	21	31	31	40

3.2 Challenges Faced by Practitioners in Variability Management

The focus group discussions mainly followed the life cycle stages of SPLE, as we mentioned in Section 2.4. However, when we were analyzing the results, we found that the majority of the issues/challenges reported by the participants are not particular to a specific phase. Therefore, we decided not to organize our reporting of the results follow the life cycle stages of SPLE. Instead, we divided them into two categories: technical issues and non-technical issues. The issues we found are summarized below.

[2] We used the European Commission SME definition: companies with 250 or more employees are considered as large size, companies with 50 (inclusive) to 250 employees are considered as medium size, and companies with 10 (inclusive) to 50 employees are considered as small size. http://ec.europa.eu/enterprise/policies/sme/files/sme_definition/sme_user_guide_en.pdf

3.2.1 Technical Issues

Handling complexity: When discussing issues in the requirements phase, the participants reported that handling the complex variability is challenging. One participant mentioned that, "when you have 300 features it is very difficult to visualize…so for us it's not easy to visualize features and show that to the customer." Maintaining complex variability models is also challenging. As one participant responded, "I mean there's no way to maintain it, I mean the maintaining, especially changing those decisions, is extremely hard because the chances that you break something is very high, because the context that you try it in is huge." The participant also commented that the research output from intelligent decision models area is far away from being applicable to the real industrial settings.

When discussing issues in the implementation phase, the participants reported that for the variability that was a bunch of numbers is relatively easy to manage, but for the variabilities that are much more complex than numbers are difficult to manage. One participant said, "but other things are much more complex, e.g., certain types of algorithms, or whatever there might be in our systems …that's not something that we know how to do very well yet…."

Knowledge harvest and management: When discussing issues in the requirements phase, several participants mentioned that a more challenging task than how to represent the variability (e.g., using feature models) is how to harvest and share the knowledge in an efficient way. They said that the technical artifacts (e.g., source code, design, and requirements specifications) are so diverse, there is no single repository where practitioners can find the required information; abstracting the requirements of different systems into a coherent and consistent variability model is challenging. Understanding the implications of different features on customers' buy-in, on the software product line architecture was also reported to be challenging.

Extracting variability from technical artifacts: The participants reported that in the situation that the software product line is built on similar pre-existing systems, extracting the variability and commonality of those systems from the various artifacts is challenging.

The same is true in the architecture phase, the participants reported that extracting the variability from technical artifacts of different similar products and building a common architecture for those products are challenging. One participant said: "We recently did [a] kind of a workshop where we took one single piece of code. Several senior architects looked at it. It took almost five or six hours just to go through one file. There were several thousands of lines of code. We tried to figure out why were these decisions made and how can we decouple and componentize this piece of software…because it is such a legacy system and there are so many variants, it's hard to tell the common places versus the points of variability".

On implementation level, the participants reported that in the situation where different similar products were developed using clone and own practices heavily, extracting the variability from the source code files is challenging. There are some clone detection tools but they have been developed mainly for single systems. If there are multiple code bases, each for one product, comparing them and extracting out variability from them is very difficult. Componentizing the existing code and building variability inside and around them are challenging.

Evolution of variability: The participants reported that in SPLE paradigm, SPLE requirements span different systems. These systems inevitably evolve over time. So the variability exists not only over space (change from system to system) but also over time (change over time). Managing the evolving requirements of SPL is challenging.

When requirements changed, in some cases, the existing architecture does not support the required variability in the new requirements. Some participants mentioned that, in an extreme case where the product line started with one single product, evolving the architecture towards a software product line is really an issue. One participant reported that initially, the company just wanted to penetrate the market with one single product without any variation. Then the product would grow over time and this also meant the architecture needed to be changed. Small changes and variation points were added gradually. After a couple of years, the architecture did not work anymore. Participants said they had not come across good solutions for such challenges. In some domains (e.g., mobile phones), the extending problem (extending the scope of product line) is evident. Some typical evolution scenarios include: adding variation points and variants, removing obsolete variation points and variants, changing relationships among variation points and variants.

Variability modeling and documentation: The participants mentioned that the variability modeling approaches are not very user friendly. How to document variabilities in a way that is easy to understand and use by different stakeholders is an issue. The participants also reported that compared to the structural aspects, managing the variability in the behavioral and timing aspects is more challenging and less solved.

Design decisions management and enforcement: The participants reported that managing architectural design decisions and enforcing those decisions are challenging. For example, one participant reported that the architecture design decisions were documented in Microsoft Word document, but they are very difficult to find by the related stakeholders. A better strategy to find these documented decisions is needed. The participant also reported that understanding how the alternative decisions can lead to different architectures, and what the implications of those decisions are on the resulting systems is a key challenge. One participant said: “understanding how to do that properly really requires a lot of experience with systems of that type actually.”

Tool support: The participants reported that there is a lack of sufficient tool support for managing variability. One participant said: “it [the tool] works pretty good for specific individual projects that you're going to deliver but for maintaining core assets it doesn’t…you know…you have to get creative as far as how to manage variability in requirements.” Two participants also reported the difficulties of integrating their chosen requirements management tool (i.e. DOORS) with currently available variability management tool. One participant said there was a tool change step from DOORS to one of the available variability management tool, so they had to develop a connection between those tools. But finally due to the fact that they cannot afford to maintain the home developed connection, it was not used. They agreed that developing and especially maintaining home grown tool is too costly. The participants expressed their expectation to have an integrated, standardized, and end-to-end tool support, instead of having different tools for closely related problems.

Testing: The participants reported that testing variability in SPL is a challenge, which can cause several problems. Large amounts of efforts are spent on software testing in industry, but they could not see much effort from researchers. For example, one participant said "If you see in practice, many people - a third or something on testing, a quarter, should be even more probably – but then if you look at the same amount of people in research […] I don't know how many percent of the researchers are really working on testing." One participant also mentioned that there is little, if any, work on testing the quality of the core asset to see how it is flexible enough to support the variations, instead of testing the products that come out of the core asset.

3.2.2 Non-technical Issues

People: During discussion of the issues in architecture phase, the participants reported that having good architects is essential for developing and maintaining a software product line architecture. However, there is a lack of good architects who are good at thinking at an abstract level, who know the mechanisms to make the architecture extensible and flexible, and can apply those principles in practice. The architects should also have a product line mindset. With this mindset, rather than architecting their new variability in a sense that it is their own product, architects are going to be architecting the new variability in a mindset that it has to fit on the platform that is already in existence without the clone and own approach.

Mindset change: The participants reported that changing employees' mindsets of building a single system to the mindset of building a family of systems is really challenging. One participant said: "the biggest problem you have really is people having the traditional mindset of building a single system and they have difficulties just talking about variation. […] just having a framework for getting people to think about things in a broader set is really the biggest challenge you have."

Management support: When discussing issues in the architecture phase, the participants reported that to keep the architecture from deterioration, sustained support from the organization (e.g. management support and required funding) is essential. However, keeping such sustained support can be challenging. These challenges generally include examining why certain managers accept and support VM, and what are the factors that can convince managers to give sustained support.

Organizational structure: The participants reported that the separation of the core asset team and the product team puts the people in the product team working in the situation where they have less choice. Getting them to accept the architecture that they did not design themselves is challenging. Proper communication mechanisms should be put into place to alleviate this issue. Some participants also reported that within their organization, barriers exist between different departments who own different yet similar products, and they do not talk to each other, which makes organizational changes very challenging.

Business model: The participants reported that smooth application of VM practices relies on a proper business model. For example, one participant said "Our business model is probably one of our biggest challenges… the specific customer we work with has a hierarchy where they don't talk to one another and so they are not incentivized to share assets and so we kind of mirror that. And so the business model is

probably the biggest challenge we have in a sense that we are paid by lines of code not how effective we share asset." Obtaining a suitable business model is challenging when the existing business model does not encourage reuse.

3.3 Frequency Analysis

During the coding process, five high level themes emerged. They are "Technical (Tech)", "Business (Biz)", "Managerial (Mng)", "Organizational (Org)", and "People". Each of these themes corresponds to one type of issues. For example, Tech corresponds to technical issues and Biz corresponds to business issues. To determine the relevant importance of these five different types of issues to the VM industrial practices, we performed a frequency analysis. Table 3 shows the results. The number in each cell represents the number of segments of speeches associated with the theme represented by the column header. It can be seen that the sum of frequency for non-technical challenges is close to technical challenges (i.e., 10+5+8+13=36, which is close to 38). These findings indicate that non-technical challenges have significant impact on VM practices. One participant also reported that: "non-technical challenges are at least as important and difficult as those technical challenges."

Table 3. Frequecies of discussion on each type of issues

Type	Biz	Mng	Org	People	Tech
Frequency	10	5	8	13	38

4 Discussion

We discuss our findings in the context of (relation to) the research output in VM research (a similar approach was used by Rabiser et al. [22]). Specifically, by research output we refer to the VM approaches proposed in the literature. Some of these approaches have been systematically reviewed from different angles and purposes and reported in our different publications [2, 5, 7-8]. The focus of the discussion in this section is to discuss the findings from the focus group in light of the findings from literature reviews we have already reported in order to highlight the issues that still remain unsolved according to the perceptions of the participants of our study.

To facilitate the discussion, we have clustered the issues into two groups based on the number of approaches reported in the literature [2, 5, 7-8]. The two groups are shown in Table 4. The first group contains the issues for which no or only few approaches have been proposed. The second group contains the issues for which several or many approaches have been proposed.

The issues in the first group identify the research areas that appear to have not been given enough attention from VM researchers despite practitioners reporting that they face these issues in their daily work and feel frustrated about not having an effective and efficient solution to address these issues.

It is obvious that the challenges belonging to the non-technical issues (i.e., people, mindset change, management support, organizational structure, and business model) category appear to have received significantly less attention from the

Table 4. The grouping of issues based on the number of approaches tried to tackle them

No/few approaches have been proposed	Several/many approaches have been proposed
– People – Mindset change – Management support – Organizational structure – Business model – Handling complexity – Knowledge harvest and management – Evolution of variability – Design decisions management and enforcement – Extracting variability from technical artifacts – Testing	– Variability modeling and documentation – Tool support

research community; none of the approaches we reviewed have tried to address any of them according to the findings from one of our literature review studies [7]. From these findings, it can be concluded that a large majority of the VM approaches nearly exclusively focus on technical aspects of VM to the extent of ignoring the VM challenges caused by non-technical factors involved such as business contexts and organizational environments. Similar observations have been made in the area of architectural description languages [17]. Encouragingly, researchers have recently begun to pay more attention to these issues by discussing them at community events [23].

So far technical challenges of VM are concerned, the issues of handling complexity, knowledge harvest and management, evolution of variability, design decisions management and enforcement, extracting variability from technical artifacts, and testing have not received sufficient attention from VM research community either. For example, extracting variability from technical artifacts has been reported as a significant challenge by the participants of our study. However, existing approaches for identifying variability (e.g., FODA [10] and DRM [19]) mainly rely on a manual process. Testing is also reported as a big challenge by the participants but there are only a few approaches for addressing testing issue exist (e.g., the ScenTED [21]). However, ScenTED does not test the core asset to see how it is flexible enough to support the variations as specially mentioned by the participants. Evolution of variability is also reported as one of the challenges by the participants; however, only three approaches, FDL [27], Ye'05 [28] and Loesch'07 [16], which are concerned with evolution of variability were found. These approaches only provide very limited support for evolution of variability, a systematic approach to provide a comprehensive support for variability evolution is not available. Handling complexity of variabilities (e.g., the huge number of variabilities and the complex relationships among them, and the complex unit of variability) has also been reported as a big challenge. Existing approaches all seem to fall short of scaling up to handle complex variability satisfactorily [6]. Encouragingly, researchers have begun to pay more attention to some of these issues by organizing community events [12] to discuss various ways of addressing the issue of handling complex variability have been held.

The issues in the second group mainly reveal/reflect the limitations of existing approaches, because although many approaches have been proposed to tackle them, practitioners are still facing these issues. In other words, the existing approaches that attempt to address those issues have several limitations, which hinder practitioners from fully utilizing (or benefiting from) them. The approaches addressing the variability modelling and documentation issue account for a large portion of the approaches we found [2, 7]. Even the standardization of variability modelling has been initiated [9]. However, as reported by the participants of our focus group, these approaches are not very user friendly. How to document variabilities in a way that is easy to understand and use by different stakeholders is still an issue. In addition, managing the variability in behavioral and timing aspects is less solved and more challenging compared to structural aspects.

Many tools have been proposed for managing variability. Even some commercial tools (e.g., Gears[3] and pure::variants[4]) have been available. However, practitioners expect to have an integrated, standardized, and end-to-end tool support, instead of having different tools for closely related problems. Such requirements have not been met, and practitioners are still suffering from this issue.

The issues in the second group call for more research to improve existing approaches or propose new approaches for satisfying practitioners' requirements.

In the above discussion, we assumed that the participants' statements on the contemporary VM issues are correct (i.e., all issues that reported in Section 3.2 and classified in Table 4 are open issues). There is a possibility that some issues that the participants reported as issues are actually not issues as practitioners may not be fully aware of the solutions to those issues proposed by the research community. We paid attention to this aspect when we were analyzing the data. We did not find such situation (based on our literature review results). This might be because the participants of our study were selected from the attendees of SPLC conference, so they tend to be aware of the recent work reported on VM research.

The findings from this study can also be used for performing comparative analysis with previously identified VM challenges such as reported in Bosch et al. [4]. Such comparison can provide useful information that would confirm the significance and relevance of the VM challenges or indicate the less importance or resolution of certain VM challenges identified many years ago. Our comparison[5] of the findings from this study and the VM issues reported by Bosch et al. [4] has revealed interesting information. Our analysis of the practitioners' views have discovered the issues that were not reported by Bosch et al. [4]; while we have also noticed that some of the issues mentioned in [4] were downplayed by the participants of our focus group. For example, many of the identified issues of non-technical nature (e.g., mindset change, management support,

[3] A tool from BigLever (http://www.biglever.com/)

[4] A tool from the pure-systems GmbH (http://www.pure-systems.com/pure_variants.49.0.html)

[5] We have performed a similar comparative analysis in which the data from this study and Bosch et al. [4] were also used. The findings from that comparative analysis have been published in [2].That comparative analysis was performed using a coding scheme defined at a higher level of abstraction than the one used for this study. Despite the data analysis efforts were led by two different researchers for [2] (i.e., the second author) and for this study (i.e., the first author), some overlaps between the findings reported in [2] and in this study, especially in the section 4 of this study, are unavoidable.

organizational structure, and business models) were not particularly emphasized in Bosch et al. [4]; nor these issues appear to have gained significant attention from other VM researchers as mentioned in Table 4. We also noticed that some of the VM issues described by Bosch et al. [4] (such as "first class representation of variability", "late binding decisions", or "stakeholders concept overlap") were either ignored or less emphasized by the participants of our focus group study. One possible interpretation of this can be either those issues are not that much important anymore or they have already been sufficiently resolved by existing VM approaches. For example, the issue of first-class representation seems to have been solved, because many approaches (e.g., COVAMOF [25] and OVM [20]) have advocated first-class citizenship of variability. However, it is difficult to make any conclusive remarks about our comparative findings because some of the VM challenges identified by others might have gone unmentioned because our sample size was too small to be expected to cover an exhaustive list of VM challenges.

In summary, the findings from this study provide useful information about and practice-based insights into the VM challenges in SPLE. These findings also not only confirm many of the VM issues reported by Bosch et al. [4] but also identify more challenges that practitioners appear to face while managing variability in SPL. Many VM challenges discovered by analyzing the perceptions of the participants of this study tend to be neglected by the VM researchers. This study has also revealed that some of VM issues reported by Bosch et al. [4] might have been solved; for example, the participants did not mention any problem in "first-class representation of variability", which was reported as a challenges by Bosch et al. [4]. The findings highlighting the importance of dealing with previously not much emphasized challenges such as "Non-technical issues" or "scalability" can be used by researchers to carve out new research agenda and directions for VM research efforts. We also expect that the results from this study will encourage researchers to carry out more studies in order to determine and understand the key problems in managing variability and the socio-technical factors that can facilitate or hinder the successful technology transfer of the outcomes of the VM research to industry.

5 Limitations

Like any empirical study, this study also has certain limitations. Our study was conducted with participants having different roles in different companies' SPL initiatives. Hence, the results are limited to the respondents' knowledge and beliefs about the challenges and issues involved in VM throughout the SPLE lifecycle. This situation can cause problems when practitioners' perceptions may be inaccurate. However, like the researchers of many studies based on opinion data (e.g., [1, 18]), we also have full confidence in our findings because we have collected data from practitioners working in quite diverse roles and directly involved in SPL activities within their organizations. Sample size may be another issue as we had only 11 participants from 11 organizations in 3 focus group sessions. To gain a broader representation of industrial challenges of VM in SPL, more practitioners and organizations need to be included in a future study. But we hope that a reader may be able to identify the challenges and some of the discussed solutions from the literature that are transferable to his/her environment. Despite the abovementioned and potentially other limitations of this empirical study, the findings from this study are expected to provide useful information about the VM issues that are perceived to be unsolved and challenging by the participants of our study.

6 Conclusions and Future Work

Effective and efficient management of variability is vital to achieve the large-scale reuse promised by the software product line paradigm. The overall goal of our research is to investigate the contemporary industrial challenges of VM in SPL after almost 20 years of research and practice. To achieve this objective, an empirical study using focus group as the data collection method was designed and executed to explore practitioners' experience and perceptions about the VM challenges in SPL.

This research has gathered empirical evidence to update and advance the knowledge about the VM challenges faced by practitioners. The findings of the study highlighted several technical issues, i.e., handling complexity, knowledge harvest and management, extracting variability from technical artifacts, evolution of variability, variability modeling and documentation, design decisions management and enforcement, tool support, and testing of artifacts with variability. Especially, the findings of the study shed light on non-technical challenges (i.e., issues regarding people, mindset change, management support, organizational structure, and business model) faced by practitioners in their daily practice of SPL. These non-technical challenges appear to have been hardly addressed by existing VM approaches, which seem to be mainly focused on technical aspects of VM [2, 5, 7-8].

The research results presented here can help researchers to identify the areas that demand further research; especially the results revealed and highlighted several neglected areas of research (e.g., tackling various non-technical challenges). Practitioners can also benefit from the findings. For example, the practitioners who are going to adopt a software product line approach can know the variability management challenges that they need to be aware of; for practitioners who have already adopted a SPL approach, the synthesized list of challenges can help them to get an understanding on what challenges their colleagues are facing, thus they can be more knowledgeable about the neglected issues in their own organizations.

Acknowledgements

We sincerely thank the participants of our focus group discussions. We gratefully thank Dr. Nour Ali for participating in the design of the sequence of questions to be asked during the focus group sessions and her help during execution of the study. We are also thankful to all other colleagues who helped us to conduct the study. The first author sincerely thanks Mr. Klaas-Jan Stol and Dr. Deepak Dhungana for reviewing early drafts of this paper. This work is supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1.

References

[1] Ali Babar, M., Bass, L., Gorton, I.: Factors Influencing Industrial Practices of Software Architecture Evaluation: An Empirical Investigation. In: Overhage, S., Szyperski, C., Reussner, R., Stafford, J.A. (eds.) QoSA 2007. LNCS, vol. 4880, pp. 90–107. Springer, Heidelberg (2008)

[2] Ali Babar, M., Chen, L., Shull, F.: Managing Variability in Software Product Lines. IEEE Software 27(3), 89–91, 94 (2010)
[3] Bachmann, F., Clements, P.: Variability in Software Product Lines, Tech. Report CMU/SEI-2005-TR-012, Software Engineering Institute, Pittsburgh, USA (2005)
[4] Bosch, J., et al.: Variability Issues in Software Product Lines. In: Software Product-Family Engineering (PFE 2001), pp. 13–21. Springer, Heidelberg (2002)
[5] Cawley, C., Chen, L., Ali Babar, M.: A systematic review of the research on variability management in software product lines, Tech. Report Lero, University of Limerick, Ireland (2008)
[6] Chen, L., Ali Babar, M.: A Survey of Scalability Aspects of Variability Modeling Approaches. In: Workshop on Scalable Modeling Techniques for Software Product Lines at SPLC 2009, San Francisco, CA, USA (2009)
[7] Chen, L., Ali Babar, M., Ali, N.: Variability Management in Software Product Lines: A Systematic Review. In: Proceedings of the 13th International Software Product Line Conference (2009)
[8] Chen, L., Ali Babar, M., Cawley, C.: A Status Report on the Evaluation of Variability Management Approaches. In: Proceedings of the 13th International Conference on Evaluation and Assessment in Software Engineering. British Computer Society (2009)
[9] Haugen, Ø.: Common Variability Language Request for Proposal – CVL RFP, http://www.omgwiki.org/variability/lib/exe/fetch.php?id=start&cache=cache&media=cvl-rfp-091209.pdf (Last accessed on March 10)
[10] Kang, K.C., et al.: Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. Report CMU/SEI-90-TR-021, SEI (1990)
[11] Kasunic, M.: Synthesis: A reuse-based software development methodology, Process Guide, Version 1.0. Tech. Report Technical Report, Software Productivity Consortium Services Corporation (1992)
[12] Kishi, T., Kang, K.-C.: Scalable Modeling Techniques for Software Product Lines (SCALE 2009) - Bridging the Gap between research and practice. In: Proceedings of the 13th International Software Product Line Conference, pp. 311–312 (2009)
[13] Kontio, J., Lehtola, L., Bragge, J.: Using the focus group method in software engineering: obtaining practitioner and user experiences. In: Proceedings of 2004 Int'l. Symposium on Empirical Software Engineering, pp. 271–280 (2004)
[14] Krippendorff, K.: Content Analysis: An Introduction to Its Methodology, 2nd edn. Sage Publications, Inc., Thousand Oaks (2003)
[15] Van der Linden, F.J., Schmid, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer, New York (2007)
[16] Loesch, F., Ploedereder, E.: Optimization of Variability in Software Product Lines. In: Proceedings of the 11th International Software Product Line Conference, pp. 151–162. IEEE, Los Alamitos (2007)
[17] Medvidovic, N., Dashofy, E.M., Taylor, R.N.: Moving architectural description from under the technology lamppost. Information and Software Technology 49(1), 12–31 (2007)
[18] Niazi, M., Wilson, D., Zowghi, D.: A framework for assisting the design of effective software process improvement implementation strategies. Journal of Systems and Software 78(2), 204–222 (2005)
[19] Park, S., Kim, M., Sugumaran, V.: A scenario, goal and feature-oriented domain analysis approach for developing software product lines. Industrial Management + Data Systems 104(4), 296–308 (2004)
[20] Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, New York (2005)

[21] Pohl, K., Metzger, A.: Software product line testing. Commun. ACM 49(12), 78–81 (2006)
[22] Rabiser, R., Grünbacher, P., Dhungana, D.: Requirements for product derivation support: Results from a systematic literature review and an expert survey. Information and Software Technology 52(3), 324–346 (2010)
[23] Schmid, K., Babar, M.A., Grünbacher, P., Nonaka, M.: The Second International Workshop on Management and Economics of Software Product Lines (MESPUL 2008), Proceedings of the 12th International Software Product Line Conference, pp. 386–386 (2008)
[24] Seaman, C.B.: Qualitative methods in empirical studies of software engineering. IEEE Transactions on Software Engineering 25(4), 557–572 (1999)
[25] Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J.: COVAMOF: A Framework for Modeling Variability in Software Product Families. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 197–213. Springer, Heidelberg (2004)
[26] Strauss, A.C., Corbin, J.: Basics of Qualitative Research: Grounded Theory Procedures and Techniques, 2nd edn. Sage Publications, Inc., Thousand Oaks (1990)
[27] van Deursen, A., de Jonge, M., Kuipers, T.: Feature-Based Product Line Instantiation Using Source-Level Packages. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 19–30. Springer, Heidelberg (2002)
[28] Ye, H., Liu, H.: Approach to modelling feature variability and dependencies in software product lines. IEE Proceedings Software 152(3), 101–109 (2005)

Consistent Product Line Configuration across File Type and Product Line Boundaries*

Christoph Elsner[1], Peter Ulbrich[2], Daniel Lohmann[2], and Wolfgang Schröder-Preikschat[2]

[1] Siemens Corporate Research & Technologies, Erlangen, Germany
[2] Friedrich-Alexander University Erlangen-Nuremberg, Germany
christoph.elsner.ext@siemens.com,
{ulbrich,lohmann,wosch}@cs.fau.de

Abstract. Creating a valid software configuration of a product line can require laborious customizations involving multiple configuration file types, such as feature models, domain-specific languages, or preprocessor defines in C header files. Using configurable off-the-shelf components causes additional complexity. Without checking of constraints across file types boundaries already at configuration time, intricate inconsistencies are likely to be introduced—resulting in product defects, which are costly to discover and resolve later on.

Up to now, at best ad-hoc solutions have been applied. To tackle this problem in a general way, we have developed an approach and a corresponding plug-in infrastructure. It allows for convenient definition and checking of constraints across configuration file types and product line boundaries. Internally, all configuration files are converted to models, facilitating the use of model-based constraint languages (e.g., OCL). Converter plug-ins for arbitrary configuration file types may be integrated and hide a large amount of complexity usually associated with modeling. We have validated our approach using a quadrotor helicopter product line comprising three sub–product-lines and four different configuration file formats. The results give evidence that our approach is practically applicable, reduces time and effort for product derivation (by avoiding repeated compiling, testing, and reconfiguration cycles), and prevents faulty software deployment.

1 Introduction and Motivation

Creating consistent configurations for larger-scale product lines often is a laborious task affecting various types of configuration files with subtle dependencies. Whereas customer-visible variability might be bound via selecting options in a feature model, the deployment of software to physical nodes may reside in domain-specific models or text files [10], while fine-tuning is done via preprocessor variables in C header files. Choosing certain features in the feature model may

* This work was partly supported by the German Research Council (DFG) under grants no. SCHR 603/4 and SCHR 603/7-1.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 181–195, 2010.

impede certain choices in the domain-specific model, while setting a preprocessor variable in turn may presuppose a feature to be set. Configuration complexity increases even further when employing configurable off-the-shelf components (e.g., Apache, Oracle) or when building combined products including other product lines, which in turn expose their variability via certain configuration file formats.

One solution often recommended and applied to ease product configuration is to explicitly limit the scope of the product portfolio by offering only a small subset of possible configurations to the customer. When strictly applied, it is possible to use a single top-level configuration file (e.g., a feature model configuration) and automatically generate all the remaining configuration files no matter which format [4]. However, a very common scenario in industry [6] is that a customer actively negotiates the characteristics of the product. This makes application engineering a laborious task requiring implementation of additional modules, but also fine-grained configuration and customization at various locations with subtle, implicit dependencies. In such a case, a predefined subset of products, configurable via a single entry point, is not realistic.

Consistency constraints that span different configuration file types need to be enforced at configuration time to prevent inconsistencies, in particular if the misconfiguration is known to manifest only sporadically during runtime (e.g., due to race conditions). One promising approach for consistency checking is to transform all configuration files into a common representation and use constraint-checking languages. Model-based development offers powerful constraint languages, such as OCL [13] or XPand Check [21], and asserts that virtually every artifact may be transformed into a model [3]. In fact, for various file types, there do exist converters or converter frameworks into the modeling world, for example, for textual grammars (XText [22]) or XML files (XMLSchema [9]). However, combining all these single frameworks in an ad-hoc manner for a specific product line in order to check constraints is a considerable amount of work. A product line engineer familiar with a particular domain (e.g., embedded systems) might not be a modeling expert as well, who is able to set up, combine, and tame all the modeling technologies and tools. Moreover, such an infrastructure should be developed in a generic way in order to apply it to various product lines. Up to now, at best ad-hoc solutions for doing constraint checking are applied in industry. As we have argued in a previous paper [8], a general approach for constraint checking across arbitrary configuration file types is missing.

To tackle this problem, we have developed an approach and corresponding tooling for an extensible constraint-checking infrastructure. It supports scenarios where multiple configuration file types are involved, which may even be spread over several sub–product-lines and off-the-shelf components. Our infrastructure provides for comfortable definition and checking of constraints independently of the configuration file type used and achieves this by transforming all configuration files to models in a transparent way. We have already developed plug-ins for several file types (such as feature models, XMLSchema, Ecore [9], KConfig, C preprocessor defines) and support different constraint definition languages (OCL, XPand Check). Arbitrary file formats may be supported—by writing converter plug-ins

for transforming configuration formats into the internally used modeling format (Eclipse Ecore). We have applied our approach to a quadrotor helicopter product line (I4Copter) comprising three sub–product-lines and four different configuration file formats. By discovering and resolving inconsistencies at configuration time, we could considerably reduce the time and effort for product derivation. Detecting them at later stages of product derivation—at compile time, testing time, or after deployment—, as it was necessary beforehand, was far more costly.

After introducing the I4Copter product line and its configurability as a concrete scenario guiding through the paper in Section 2, we contribute a practice-oriented approach for mapping configuration files of different types to models (Section 3). Then, we account for our generic, extensible constraint-checking framework, which provides comfortable means for constraint definition while minimizing the necessary knowledge about model-based development for the product line developer and the configuration engineer (Section 4). Sections 5 and 6 report on its application to the I4Copter and the achieved results. Finally, we discuss our approach and related work (Sections 7 and 8).

2 Scenario: Quadrotor Helicopter Product Line

The evaluation platform for our product line research is the *I4Copter* [19] quadrotor helicopter, which has been designed and developed to resemble embedded real-time systems arising in real-world product line scenarios. The I4Copter software product line comprises three sub–product-lines: one product line for application logic, which also models the interface to the hardware (*CopterSwHw*), the commercial operating system product line *PXROS*[1], and, as an alternative operating system product line, the department-internal *CiAO* [11].

The *CopterSwHw* product line is implemented in C++; various software features are optional and alternative (approximately 50 features in total). There are several hardware variants of the I4Copter, comprising different frames, sensors, and actuators, so that the hardware as well forms a product line. The software depends on information about the available hardware components (approximately 60 features). Both the software part and the available hardware components of the *CopterSwHw* product line are configured via C preprocessor directives.

The application logic may run either on the operating system product line *PXROS* or on *CiAO*. For configuring PXROS (tasks and other parameters), a simple textual domain-specific language (DSL) is used, out of which a generator creates the corresponding C start-up code. While PXROS is more reliable and mature, the department-internal operating system product line CiAO is much more versatile. It uses pure::variants [2] feature models for configuring various functional and architectural properties and comprises an own XML dialect, defined in XMLSchema, for configuration of operating system tasks.

Domain-specific configuration constraints of the I4Copter span several files and are therefore hard to enforce (see also Figure 1). When equipping the hardware with an acceleration sensor using the Serial Peripheral Interface (SPI) bus,

[1] HighTec EDV Systeme GmbH - http://www.hightec-rt.com/

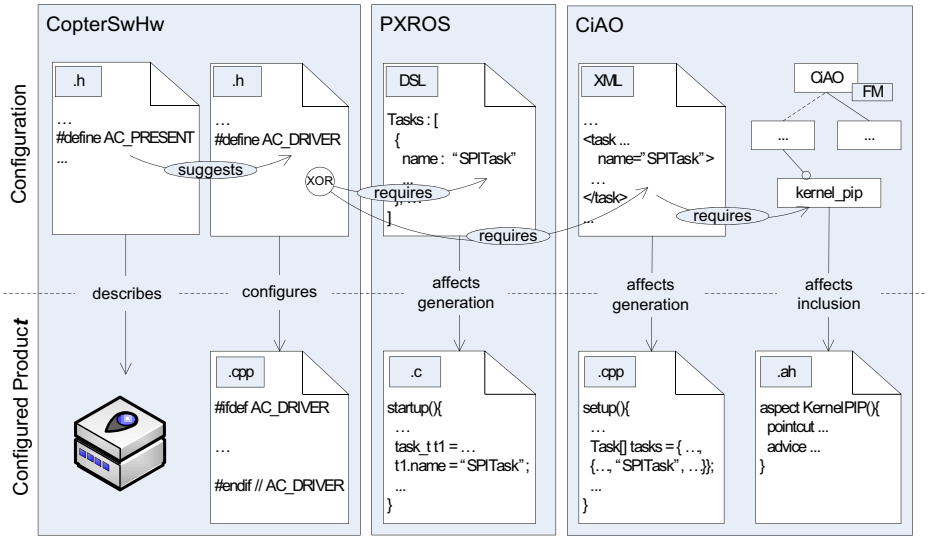


Fig. 1. Exemplary domain constraints within the I4Copter product line

for example, this has to be described in the corresponding hardware header file (`AC_PRESENT`). The application software product line usually would (although not necessarily!) be configured with the corresponding sensor device driver in the software header file (`AC_DRIVER`). Using any SPI device requires that an SPI bus controller software module is present, which is implemented as an operating system task. Thus, a corresponding operating system task (`SPITask`) needs to be initialized appropriately either by using the DSL of PXROS or the XML language of CiAO. When choosing CiAO as the operating system, further configuration details need to be enforced, as CiAO is highly configurable. When including the SPITask, it must be ensured that it sends its messages to the bus with the same priority as the task that triggered the message. This requires selecting an appropriate priority mechanism in the feature model configuration, for example the priority inheritance mechanism (`kernel_pip`).

There exist various of such domain-specific constraints in the I4Copter product line. Some are rather recommendations or warnings, others are mandatory. The constraints span several different configuration file types and product lines. This makes the configuration of the I4Copter product line sufficiently complex to resemble the challenges faced in industrial-scale embedded systems development. We will return to the *I4Copter* as an illustrative example when we explain the general approach, its architecture, and validation.

3 Product Line Configuration and Modeling

Our approach is based on the assumption that every artifact involved in product line configuration has a corresponding representation in the modeling world [3].

Basically, the configuration of a concrete product is described by its *set of configuration files*. The product line itself defines, either implicitly or explicitly, the *set of configuration file types* it can deal with. As mapping of product lines and configurations into the modeling world is not without pitfalls, we will now first introduce general modeling terminology and then explain details how we perform the mapping of configuration files and configuration file types.

3.1 Modeling Terminology

A *model* is a formal abstraction of a concept (e.g., a physical system or software) describing its concrete entities and relationships [16]. A model is abstract in the way that it is not tied to a certain textual or graphical representation. The formal rules, which specify the entity and relationship *types* allowed in a certain model, are provided by its *metamodel*. As an example, a simple metamodel for modeling operating system tasks will comprise the root entity type `TaskList`, which contains an arbitrary number of `Task` elements, which in turn have a `name` element of type string and a `priority` of type integer. A model conforming to this metamodel, for instance, defines a concrete TaskList comprising exactly one Task with `name = "SPITask"` and `priority = 1`. For specifying metamodels, a *metamodeling technology* is used (e.g., our implementation uses Eclipse Ecore).

Within one metamodeling technology, it is possible to define constraints, such as the SPI bus constraints described in Section 2, in a formal, machine-interpretable way. Common examples for such languages are OCL and XPand Check. Constraints on models usually are specified using the elements defined in the metamodel. For example, a constraint on a task model can leverage the fact that each TaskList has a number of Tasks, which in turn have a name and a priority. Querying whether there is any Task called "SPITask" for the whole system can therefore be formulated very concisely, for example in XPand Check, which implicitly iterates over sets: `myTaskList.tasks.name.contains("SPITask")`. Subsequently, we will describe how we map product lines and their configurations into the modeling world, that is, to metamodels and models.

3.2 Mapping Product Line Configurations to Models

A configuration file is an artifact that specifies the characteristics of a product (e.g., a web-server configuration file, C header files, but also domain-specific model and text files). A *configuration file* therefore corresponds to a *model*. The elements and the relations allowed in the configuration file (i.e., the abstract syntax part of the *configuration file type* exempt from its concrete syntax) correspond to what is the *metamodel* of this model. Accordingly, a *configuration* of a product line can be seen as a *set of models*, the *product line* itself as a *set of metamodels*. Although the mapping is straight-forward, it is still open how to actually *derive* metamodels from a product line.

We distinguish two classes of configuration file types, which differ in the way to derive their metamodel. Firstly, there are those for which the product line

already provides an explicit specification file that can be converted to a *product-line–specific metamodel*. Secondly, there are those where this is not the case, and only a less expressive, *generic metamodel* can be used.

Product-Line–Specific Metamodels via Specification File. Some product lines comprise an explicit specification of what is a valid configuration file for them. For example, the CiAO product line defines the format for a valid task configuration XML file in XMLSchema. For PXROS, we developed a simple textual grammar specifying which constructs are valid in its domain-specific configuration file. Feature models can be interpreted as metamodels as well [17]. Finally, model-driven product lines specify their metamodels explicitly (e.g., [20]). Tools that map specification files to *product-line–specific metamodels* are already available in many cases, for instance, for XMLSchema files and for XText grammars there exist converters that derive the corresponding Ecore metamodels [9,22].

Generic Metamodels Without Specification File. There are, however, also configuration files that lack any expressive and formal specification of what the valid constructs are in the context of one particular product line. This is, for example, the case for Java property files and for C header files containing preprocessor defines. In principle, arbitrary identifiers may be set to arbitrary values in a preprocessor define statement, such as `#define AC_PRESENT 1`. Which defines are necessarily required, optional, or unused, or what the permissible value ranges are for a particular product line, is not specified explicitly and can only be discovered by reading source code or documentation. Although a product line engineer could use this information to reconstruct a product-line–specific specification file (e.g., an XText grammar), this often will not be the case.

We therefore see the need to map certain file types to less expressive *generic metamodels*. For a C header file with preprocessor defines, for example, such a metamodel will only specify that a `DefineList` contains an arbitrary number of `DefineStatements` having an `identifier` of type string and a `value` of type string. This fact makes defining explicit constraints more chatty and error-prone. For example, having a specific metamodel, a constraint on the debug-level define may be formulated very concisely: `copterSwHw.debugLevel == 1`. Having only a generic metamodel, one needs to query the define value in a reflective way: `copterSwHw.getPropByName("debugLevel").toInteger() == 1`. However, the simplicity of such a generic metamodel also has one benefit: it can be reused across product line boundaries very easily.

Wrap-Up. To sum up, it is possible to map all configuration file types to metamodels and the corresponding configuration files to models. Tooling for conversion either exists or may be developed. However, combining all these single technologies and tools in an ad-hoc manner for a specific product line in order to check constraints is not appropriate. We will therefore present a corresponding generic tool framework in the next section.

4 Product Line Constraint-Checking Framework

In this section, we present the *PLiC* (*P*roduct *Li*ne *C*onfiguration) framework, which allows for constraint-checking across product line boundaries and configuration file types, while minimizing the required modeling knowledge of product line and configuration engineers. The PLiC framework is implemented as an Eclipse extension and enriches the integrated development environment with a `builder` component, which performs model conversion and validation in the background when a configuration artifact changes within the workspace. Both model `converters` (for specific configuration file types) and model `validators` (for evaluating constraints of different checking languages) have been implemented using the Eclipse extension point mechanism. Thus, additional configuration file types and constraint-checking languages can be implemented and integrated easily.

We will describe the builder, the converters and validators that are already available, and the extensibility of the PLiC framework later in this section. First, we address the concepts that end users have to deal with.

4.1 End User View on the PLiC Framework

There are two roles of end users: *product line engineer* and *configuration engineer* (cf. Figure 2). The former declares the set of possible configuration file types in a so called *PLiCFacade* model (step 1) and implements the domain constraints (step 2). The configuration engineer, in turn, creates the *PLiCInstance* model (step 3), which specifies the locations of the configuration files. The PLiC framework than permanently enforces the constraints on the configuration (step 4).

The PLiCFacade model. Within the *PLiCFacade* model (cf. Figure 3), the product line engineer declares three distinct sets of elements: the plug-ins required (*PLiCPlugins*), the configuration file types the product line can deal with (*ConfigFileTypes*), and references to other sub–product-lines (*PLiCFacadeRefs*). A *PLiCPlugin* has a unique ID to identify the plug-ins installed in the workbench. It may either be a *Validator* or a *Converter*. A *Validator* is parameterized with the path to a file (or directory) containing the set of constraints to check. *Converters*, finally, exist in two flavors. *SpecificConverters* (in analogy

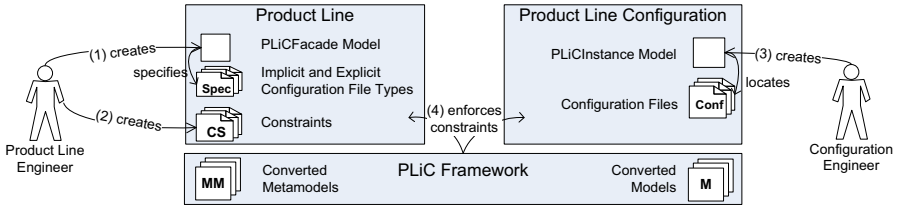


Fig. 2. Product line engineer and configuration engineer using the PLiC framework

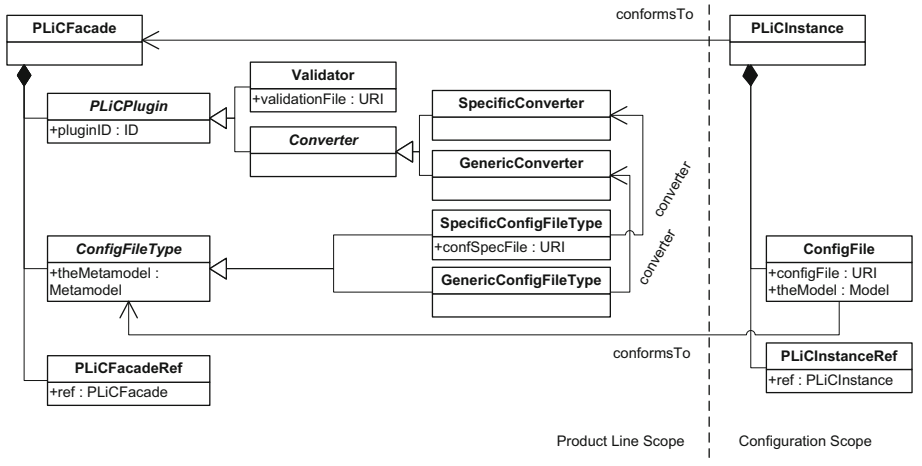


Fig. 3. First, the product line engineer defines a *PLiCFacade* model for each product line. It declares the plug-ins used, the configuration file types available, and references to other PLiCFacades. The configuration engineer, in turn, defines the corresponding *PLiCInstance* model. It specifies the locations of the actual configuration files and references to other PLiCInstances.

to Section 3.2) convert a specification file to a product-line–specific metamodel. A *GenericConverter* simply provides a single, nonspecific metamodel (e.g., a C preprocessor define metamodel), that can be used in various product lines.

Accordingly, *ConfigFileTypes* are either specific or generic. In case of a *SpecificConfigFileType*, the engineer needs to provide the URI to a specification file (XMLSchema, grammar, etc.) that may be transformed via the referenced *SpecificConverter*. For *GenericConfigFileTypes*, this is not necessary as the metamodel of the referenced *GenericConverter* is generic and fixed.

The PLiCInstance model. In the *PLiCInstance* model, the configuration engineer first references the corresponding *PLiCFacade* model it conforms to. Then he declares all configuration files and their URIs within the file system (*ConfigFiles*). Each *ConfigFile* needs to reference the *ConfigFileType* it corresponds to. Finally, the configuration engineer draws the references to other used *PLiCInstances* (sub–product-line configurations).

4.2 Builder

The `builder` plug-in works in the background and is invoked on each change in the workspace. It keeps the created metamodels and models up to date, invokes the constraint checks, and displays the check results. It builds all projects in a linear order according to their project dependencies.

The builder achieves the actual conversion to metamodels and models by invoking the appropriate `converter` plug-ins. It stores the metamodels and models

in the file system and sets references to them using the *theMetamodel* and *theModel* member variables of all *ConfigFileTypes* and *ConfigFiles* (cf. Figure 3). Doing so, the PLiCFacade and PLiCInstance models provide single entry points for performing constraint checks. Eventually, the constraints checks referenced in the PLiCFacade model (via *Validator* elements) are evaluated on the PLiCInstance model calling the appropriate `validator` plug-ins. Subsequently, we will describe the development of converter and validator plug-ins.

4.3 Converters

`Converter` plug-ins perform the actual conversion to metamodels and models and need to implement certain interfaces. A generic converter needs to provide one method for querying its fixed metamodel and one for converting a configuration file to a model conforming to the metamodel. A specific converter works similar, however has, as an additional parameter, in each of its methods the specification file, which defines the product-line–specific metamodel.

Up to now, we have developed two generic converters and five specific converters. The generic converters comprise a converter for files containing statements of the form `#define <ID> <Value>` as output by the GNU compiler `gcc` when invoked over the source code with certain parameters. Furthermore, we developed a converter for property-value files, as common for Java.

The specific converters create metamodels for specification files in Ecore, XMLSchema, and XText textual grammars. EMF already provides Java APIs for this purpose, whose complexity is hidden behind the lean converter interfaces. Furthermore, we developed a specific converter for pure::variants feature models. We intentionally keep the metamodel generated from a feature model simple. In particular, we ignore any hierarchical or dependency information of features and only create one metamodel element for each feature, having the same attributes as the feature. Actually, this is the only information necessary for constraint checking, as the configuration editor of pure::variants will ensure that constraints defined *within* the feature model itself are adhered to. Finally, we developed a specific converter that generates a metamodel from KConfig files, which are used, for example, to specify the configuration options of the Linux kernel.

4.4 Validators

A `validator` plug-in evaluates the constraints for a certain constraint definition language. The corresponding interface comprises only one method, which receives the PLiCInstance object to check and the file containing the constraint rules and returns detailed information on warnings and errors that occurred during validation of the models. Note that the builder (cf. Section 4.2) has enriched the PLiCInstance object by setting the *theModel* property of each *ConfFile* element to the built models. Doing so, the PLiCInstance provides a single entry point to all generated models possibly subject to checking.

Currently, we provide two validator plug-ins, one for defining constraints in the OMG Object Constraint Language (OCL) [13] and one for the XPand Check

language [21]. Examples for constraints will follow in the next section, where we will present the application of the PLiC framework to the I4Copter.

5 Application Scenario: I4Copter

We have evaluated the approach and the PLiC framework with the I4Copter product line described in Section 2. Six steps need to be performed in general for any product line: (1) identify configuration file types, (2) select or develop converters, (3) create PLiCFacade projects and models, (4) define initial constraints, (5) configure products using PLiCInstance projects and models, and (6) constantly maintain and improve constraints during product line evolution.

1. Identify configuration file types. The first step is to identify the configuration file types of each involved product line and configurable component. Section 2 already comprises this task for the I4Copter. It comprises a hardware product line tangled with an application logic product line, both configured via header files. The operating system PXROS is configured via a configuration file DSL, while the operating system product line CiAO uses an XML dialect defined in XMLSchema and a pure::variants feature model for configuration.

2. Select or develop converters. The initially developed converters (cf. Section 4.3) resemble the needs of the I4Copter product line. We use the preprocessor-based generic converter for extracting models from C header files. For converting PXROS' DSL configuration files, we have developed a simple grammar (less than 20 rules) for XText, which we use to derive a corresponding metamodel and a parser for converting a configuration file into a model. For the task configuration file specified in XMLSchema and the pure::variants feature model, we use the corresponding specific converters as well.

3. Create *PLiCFacade* projects and models. At this point, the product line engineer creates a new Eclipse PLiCFacade project (or converts the existing project) for each product line. Each project needs to provide exactly one PLiCFacade model file describing the required converter and validator plug-ins, the configuration file types, and possibly the location of corresponding specification files, as well as references to sub-facades.

4. Define initial constraints. For constraint definition, the product line engineer chooses (or possibly develops) validator plug-ins for the checking languages to use. For I4Copter, we only make use of the XPand Check language, as the tooling support in Eclipse is far better than for OCL. In particular, the XPand editor has an excellent code completion facility, which can be configured to load all generated metamodels. This eases the definition of constraints considerably. Furthermore, we generate additional helper functions for easy navigation through PLiCInstances and generated models. Thus, we can query, for example, the maximum priority of tasks defined in the CiAO feature model from the top level I4Copter product line configuration using the following string leveraging code completion support: `ciAOInstance().fmConf().kernel.maxTaskPriority`.

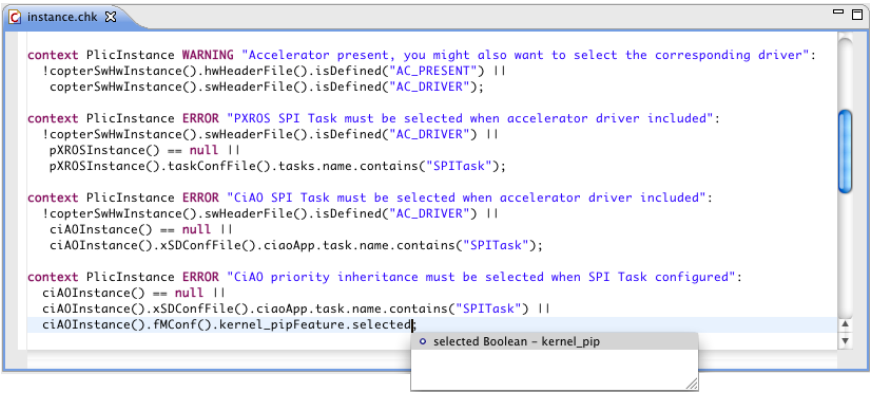


Fig. 4. Using the PLiC framework we specified the configuration dependencies of an acceleration sensor in the XPand Check constraint language

Figure 4 shows the XPand Check code necessary to encode the constraints regarding the selection of a sensor using the SPI bus as motivated textually in Section 2. The example constrains span define files, XML and DSL files, as well as feature model configurations. Note that, for a consistent configuration, each constraint needs to evaluate to `true`. By initially interviewing the I4Copter experts, we were able to find various other obligatory and recommending constraints.

5. Configure products using *PLiCInstance* projects and models. Having defined the PLiCFacade and the initial constraints, the product line configuration engineer can start configuring a product. This works via creating a new PLiCInstance project in Eclipse and filling its PLiCInstance model, which basically contains the locations of configuration files in the file system. The builder component now constantly observes the configuration files, converts them to models on each change and checks the constraints defined in the previous phase in a background process. If constraints evaluate to false, the textual messages associated with them (the error and warning strings in Figure 4) are displayed in the Eclipse problems view, and the configuration engineer gets immediate feedback and advice.

6. Constantly maintain and improve constraints during evolution. The more domain knowledge is encoded in formal constraints, the more powerful our approach is. When the product line engineer constantly maintains and improves the set of constraints, and the configuration engineer contributes as well with constraints gathered during configuration creation, our framework can give helpful guidance and considerably shortens time and cost of product configuration.

6 Results

Introducing our approach to the I4Copter was little effort. Setting up the PLiC framework and performing an initial workshop for constraint mining and

formulation took less than a day. While, during the workshop itself, only few constraints where actually defined, the I4Copter engineers got a feeling for the potential of the approach and delivered several dozens of constraints in pseudo code within the following days. We translated the pseudo code into XPand Check, and—via learning by example—the I4Copter experts rapidly grasped the relevant concepts and formulated the constraints in XPand Check themselves.

The recent introduction, however, impedes giving exact numbers on the achieved improvements. Furthermore, the success of our approach relies on various interdependent factors, such as the complexity of the product line, the quality and number of constraints, and the duration and rate of product derivations, so individual results are hardly transferable. We can, however, give anecdotal evidence that led configuration engineers to the estimation that derivation time (start of configuration to successful deployment) could be reduced by half with the currently defined constraints. Each avoided compiler run due to a triggered constraint saves up to three minutes, software unit testing ten minutes, avoiding software and hardware testing in the testbed saves a test engineer several hours.

Locating the actual source of a configuration error can be even more time-consuming. For example, choosing the SPITask but not the priority inheritance mechanism in CiAO resulted in intermittent errors such as sporadically missing of deadlines and even fatal deadlocks for the quadrotor system. Detecting and tracing back this behavior was a thankless, time-consuming task. Although our constraint infrastructure could not prevent this misconfiguration when it happened for the first time, the hereupon encoded constraint now directly triggers at configuration time and gives helpful advice for correction, so that this configuration error can no longer happen.

7 Discussion

In the following, we address several threats to the general applicability of our approach. Possible issues are the problem of semantic loss when converting among metamodeling technologies, the modeling expertise required for our approach, constraint evolution, and the relationship to generative product line approaches.

Semantic Loss. In principle, any formal, parsable file can be translated into a model and its formal specification into a metamodel. However, even if a converter is written very carefully, there will usually be semantic loss. UML's MOF, for example, is far more expressive than Ecore, which we use. But, for the purpose of constraint checking, not all this semantic information is actually needed. One can even intentionally keep a metamodel simple to simplify checking. As mentioned, our metamodel converter for feature models does neither preserve the hierarchy of features nor their dependencies. As the corresponding feature model configuration tool already enforces these dependencies, converting and checking them a second time would be unnecessary complicating and redundant.

Required Modeling Expertise. The configuration via PLiCFacade and PLiCInstance models appears straight-forward to us, and, if desired, the same

information could also be entered via plain text files or a graphical user interface. So, the only point where modeling expertise is actually needed to a certain extent is for defining constraints. A product line engineer (but not the configuration engineer) has to learn the corresponding constraint language. However, as argued in Section 5, languages such as XPand Check come with mature, content-assisting editors minimizing the necessary learning efforts.

Constraint Evolution. If not maintained, the formal constraints defined with our infrastructure will go out of sync with the actual dependencies within the software over time. This is, however, a general problem when defining dependencies, and needs to be ensured for dependencies within features in feature models, as well as for dependencies between #defines (such as, `#ifdef DEBUG #define USE_SIMULATED_SENSORS ... #endif`). Thorough development processes including, for example, code reviews and configuration testing, may assist in keeping the set of constraints consistent.

Generative vs. Constraining Approach. Using constraints for creating valid configurations does not oppose to using generative technologies. Generating source code or some of the configuration files that the software requires is often essential for efficient product derivation. We see our approach as a complement to the generative strategy in three cases: First, when fine-tuning at various location becomes necessary that cannot be anticipated beforehand, second, when multiple, orthogonal configuration files are needed that cannot be created out of one another, and, third, when introducing a single top-level configuration file that basically mirrors all configuration options on the lower levels does not scale.

8 Related Work

Related work can be found both in the field of constraint checking and in large-scale product line configuration.

Considerable research has been conducted with respect to constraint enforcement involving different types of product line models. In [5], OCL constraints ensure that UML models enriched with feature templates result in well-formed models when parameterized with valid feature model configurations. The authors of [12] relate feature models and orthogonal variability models to each other to facilitate automated reasoning on variability. Commonly, existing approaches restrain their focus on few dedicated model types and do not focus on building a general infrastructure. The FAMA tool suite [1] is a notable exception. It also provides an infrastructure for importing models and performing analyses on them. In contrast to our approach, the internally used modeling format is not based on generic metamodeling but on feature modeling. Whereas this considerably limits the types of (configuration) files and models that can be imported, the framework provides means for performing more stringent analyses, for example, for satisfiability via SAT solving. It would be very interesting to integrate validators of this kind also into our infrastructure for checking properties on the corresponding subset of model types.

Currently, we evaluate all constraints each time a file is changed and its model is rebuilt. There is research on efficient algorithms regarding incremental

consistency checking, where only those constraints are reevaluated that actually may be affected by a change [7]. Up to now, we have not run into any performance problems, the builder background process finishes in less than a second for building all metamodels, models, and evaluating the constraints of the I4Copter.

There are several approaches that deal with multiple product lines and their configuration. The Koala approach, which is applied in industry, provides for configuration of large-scale product lines via an architectural description language [14]. Approaches stemming from research, for example, use a combination of class and feature modeling [15] or service-oriented abstraction [18] to configure product lines. One of the main features of our approach is that it is agnostic to the configuration file type used and can check constraints among arbitrary configuration files as long as there exists a converter for this.

9 Conclusion and Outlook

With this paper, we have presented an approach and an infrastructure for constraint checking across configuration file types and product line boundaries. We have evaluated our approach using the *I4Copter* product line, which yielded promising results for its applicability in similar complex contexts.

As future work, we consider blurring the boundary between constraint-based and generative strategies by not only checking constraints, but also actually changing configuration values according to values in other configuration files. This, however, requires further analyses and tooling. On the one side, it is necessary to identify the cases where automatic changing of configuration values is reasonable and comprehensible for the configuring engineer, and in which cases it would rather lead to unforeseen side effects. Furthermore, this approach requires converting the changes on model level back to the source configuration files. This back-transformation is far more intricate and we need to analyze the circumstances under which this can successfully be done.

Acknowledgments. We thank Martin Hoffmann for providing insights and defining various domain constraints within the I4Copter product line and Wanja Hofer and Christa Schwanninger for their valuable feedback on this paper.

References

1. Benavides, D., Segura, S., Trinidad, P., Ruiz-Corts, A.: FAMA: Tooling a framework for the automated analysis of feature models. In: 1st Int. W'shop on Variability Modelling of Software-Intensive Systems (VAMOS) (2007)
2. Beuche, D.: Variant management with pure::variants. Tech. rep., puresystems GmbH (2006), http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf% (visited 2009-03-26)
3. Bezivin, J.: On the unification power of models. Software and Systems Modeling 4(2), 171–188 (2005)

4. Czarnecki, K., Eisenecker, U.W.: Generative Programming. In: Methods, Tools and Applications, AW (May 2000)
5. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness OCL constraints. In: 6th Int. Conf. on Generative Programming and Component Engineering (GPCE 2006), pp. 211–220. ACM, New York (2006)
6. Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software product families: a case study. Journal of Systems and Software 74(2), 173–194 (2005)
7. Egyed, A.: Scalable consistency checking between diagrams - the ViewIntegra approach. In: 16th IEEE Int. Conf. on Automated Software Engineering (ASE 2003). IEEE Control Systems Magazine, Washington (2001)
8. Elsner, C., Lohmann, D., Schröder-Preikschat, W.: Product derivation for solutiondriven product line engineering. In: 1st W'shop on Feature-Oriented Software Development (FOSD 2009). ACM, New York (2009)
9. Eclipse modeling framework homepage, http://www.eclipse.org/emf/ (visited 2010-02-22)
10. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons, New Jersey (2008)
11. Lohmann, D., Hofer, W., Schröder-Preikschat, W., Streicher, J., Spinczyk, O.: CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In: 2009 USENIX TC, USENIX, Berkeley, pp. 215–228 (June 2009)
12. Metzger, A., Heymans, P., Pohl, K., Schobbens, P.Y., Saval, G.: Disambiguating the documentation of variability in software product lines. In: 15th IEEE Int. Conf. on Requirements Engineering (RE 2007), pp. 243–253. IEEE Computer Society, Washington (2007)
13. Object Management Group (OMG): Object constraint language, version 2.2. formal/2010-02-01 (February 2010)
14. van Ommering, R.: Building product populations with software components. In: 24th Int. Conf. on Software Engineering (ICSE 2002), pp. 255–265. ACM, New York (2002)
15. Rosenmller, M., Siegmund, N.: Automating the configuration of multi software product lines. In: 4th Int. W'shop on Variability Modelling of Software-intensive Systems (VAMOS) (January 2010)
16. Stahl, T., Völter, M.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, Chichester (2006)
17. Stephan, M., Antkiewicz, M.: Ecore.fmp: A tool for editing and instantiating class models as feature models. Tech. rep., University of Waterloo, 200 University Avenue West Waterloo, Ontario, Canada (August 2008)
18. Trujillo, S., Kästner, C., Apel, S.: Product lines that supply other product lines: A service-oriented approach. In: First Workshop on Service-Oriented Architectures and Product Lines. Special Report. CMU/SEI-2008-SR-006 (September 2007)
19. Ulbrich, P.: The I4Copter project – Research platform for embedded and safety-critical system software, http://www4.informatik.uni-erlangen.de/Research/I4Copter/ (visited 2010-02-22)
20. Völter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software development. In: 11th Software Product Line Conf. (SPLC 2007), pp. 233–242 (2007)
21. Eclipse XPand homepage, http://www.eclipse.org/modeling/m2t/?project=xpand (visited 2010-02-22)
22. Eclipse XText homepage, http://www.eclipse.org/Xtext/ (visited 2010-02-22)

Automated Incremental Pairwise Testing of Software Product Lines

Sebastian Oster[1], Florian Markert[2], and Philipp Ritter[1]

[1] Real-Time Systems Group
[2] Computer Systems Group
Technische Universität Darmstadt, Germany
{sebastian.oster,philipp.ritter}@es.tu-darmstadt.de,
markert@rs.tu-darmstadt.de

Abstract. Testing Software Product Lines is very challenging due to a high degree of variability leading to an enormous number of possible products. The vast majority of today's testing approaches for Software Product Lines validate products individually using different kinds of reuse techniques for testing. Due to the enormous number of possible products, individual product testing becomes more and more unfeasible. Combinatorial testing offers one possibility to test a subset of all possible products. In this contribution we provide a detailed description of a methodology to apply combinatorial testing to a feature model of a Software Product Line. We combine graph transformation, combinatorial testing, and forward checking for that purpose. Additionally, our approach considers predefined sets of products.

1 Introduction

Various domains already apply Software Product Line engineering successfully to overcome the well-known needs of the Software Engineering community like increasing quality, saving costs for development and maintenance, and decreasing time-to-market [1]. Software Product Lines (SPLs) offer a systematic reuse of software artifacts within a range of products sharing a common set of features. The concept of Product Lines is not new and engineers in various domains like the automotive sector have adopted this concept of development for the last decades to benefit from the advantages mentioned above. Due to the variability and the systematic reuse of software components in rather different combinations and contexts, testing SPLs is very challenging.

One of the most common approaches for SPL testing is to test each product derived from the SPL individually. In the automotive domain we are running into a situation where each car of a certain brand has an individual software configuration. Furthermore, test personnel often has a certain time period to execute tests and the question arises what should/can be tested during that period of time. Engineers from the Software Engineering Community and from various industrial domains are seeking methodologies to reduce the effort of testing SPLs. Thus, it seems to be promising to have a look at lessons learned in

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 196–210, 2010.

the field of test case reduction based on parameterization. Combinatorial testing and especially pairwise testing are well-known approaches in that category.

1.1 Contribution

The contribution of this paper is the introduction of a methodology to apply pairwise testing to SPLs. This is done by translating the feature model into a binary constraint solving problem (CSP). We then generate a set of valid products containing all valid pairs of features. Testing this set of products is equivalent to pairwise testing the whole SPL. The developed pairwise algorithm can handle dependencies between features and guarantees the generation of valid products containing all valid pairs of features. Additionally, our pairwise algorithm can process existing products of the SPL ensuring that the pre-selected products are part of the product set realizing pairwise coverage. Our algorithm then generates products covering all remaining pairs which are not covered by the pre-selected products. Our approach is presented by means of a cell phone SPL which serves as a running example. We also briefly introduce our **Mo**del-based **So**ftware **Pro**duct **Li**ne **Te**sting framework (MoSo-PoLiTe) which combines pairwise testing and provides a concept to derive test cases for the generated set of products utilizing model-based testing.

1.2 Outline

The remainder of this paper is organized as follows: In the next section we describe the fundamentals relevant for our approach. In Section 3 we then introduce our Model-Based Product Line Testing framework (MoSo-PoLiTe) which realizes our combinatorial/pairwise test approach. Furthermore, we describe our pairwise testing methodology and its integration with the configuration management tool pure::variants. Afterwards, we validate our implementation in Section 4. Section 5 summarizes related work especially focusing on SPL-testing and feature model transformation. Finally, Section 6 summarizes and concludes our paper and gives an overview of future work.

2 Fundamentals

To depict the functionality of our approach we introduce our running example—a cell phone SPL based on the Google Android platform. The Android platform offers a wide variety of functionalities. We use a mixture of existing and self-developed features in our cell phone SPL. In the remainder of this paper we use a small subset of our cell phone SPL comprising the following functionalities:

Basic Functions: Making voice calls and sending messages belong to the basic functionalities of our running example.

Communication: A product of our Cell Phone SPL may contain Bluetooth, WLAN, and UMTS for additional communication purposes

Extras: An mp3 player or a camera may be part of a cell phone.

2.1 Software Product Lines and Feature Modeling

In SPL engineering modular software components are reused within a specific problem domain [1,2]. Feature models (FMs) offer an explicit representation of the commonalities and variabilities in an SPL [3]. To satisfy this purpose an FM consists of features which represent logical groups of requirements [4] or, as defined in [5], "a system property that is relevant to some stakeholder". FMs are by themselves insufficient for a complete modeling of an SPL and are usually supported by development artifacts such as code fragments, UML diagrams, or function network diagrams that are traced to the corresponding features.

Kang et al. introduced the first FMs comprising mandatory, optional, and alternative features as part of the FODA feasibility study in 1990 [6]. These feature properties are called *node notations*. Additionally, it is possible to use textual require and exclude constraints between features. Those cross-tree dependencies together with the node notation and the hierarchical structure determine, which combinations of features are permitted to create a product of the SPL [7]. A valid product is always a valid subtree of the corresponding FM. Since the introduction of FODA, further extensions of FMs were introduced to improve precision and expressiveness. We use a notation similar to the FODA FM with an additional or-group and graphical cross-tree dependencies. Fig. 1 depicts the FM of our running example.

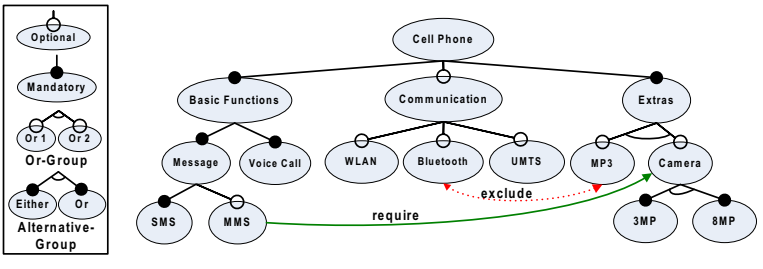


Fig. 1. Feature model of our running example

The features **Basic Functions**, **Messages**, **Voice Call**, and **SMS** are mandatory and part of every product derived from the cell phone SPL. The feature **MMS** is optional for product instantiation. **Communication** and its subfeatures: **WLAN**, **Bluetooth**, and **UMTS** are optional as well. The feature **Extras** is mandatory and the underlying or-group prescribe that at least one element of the or-group (**MP3** or **Camera**) has to be selected. It is also possible to select both **MP3** and **Camera** within the same product. Either the **3MP** (3 megapixel) or the **8MP** (8 megapixel) has to be chosen if **Camera** is included. Considering the notation and the constraints, 61 valid products can be derived on the basis of this FM. The FM was added to the FM repository on the SPL Online Tool website [8] in order to provide it to the community.

2.2 Combinatorial Testing

Combinatorial testing is a popular method to decrease the effort for testing single systems by reducing the amount of test cases. To prove the correctness of a program, it needs to be tested with all combinations of possible input parameter values [9]. Due to the complexity and size of the majority of products, testing all possible combinations of input parameter values is not feasible. A software with five different input parameters where each parameter can be initialized in 10 different ways would require $10^5 = 100000$ different test cases to be validated. Cohen et al. summarize existing approaches for combinatorial testing distinguishing between mathematical, greedy, and meta-heuristic approaches in [10]. One of the best-known applications of combinatorial testing is the pairwise testing approach. This method is based on the assumption that the majority of faults originate from a single parameter value or are caused by the interaction of two values [11]. Therefore, pairwise testing generates all possible combinations of pairs of input parameters. Many algorithms realizing pairwise testing exist; amongst others AETG [12] and IPO [13] are frequently discussed. Both achieve a 100% pairwise coverage and result in a very small set of test cases. Therefore, we implemented a pairwise algorithm similar to those approaches and added some additional functionalities to apply it to SPLs.

3 Our Approach — MoSo-PoLiTe

The MoSo-PoLiTe framework provides a test framework for SPLs and was initially developed during the feasiPLe BMBF project [14]. Fig. 2 depicts a rough overview of the MoSo-PoLiTe testing process. The central component in MoSo-PoLiTe is the FM of the SPL. It is created on the basis of the SPL requirements during domain engineering. Since the FM only provides a hierarchical overview of variable and common functionalities within the SPL it is traced to further requirement documentation as described by Wübbeke in [15]. For a model-based test case derivation it is additionally traced to a test model representing the behavior of the SPL. The frequently used test model is based on the model-based testing approach of Weissleder et al. [16]. However, all model-based approaches for SPLs like ScenTED [17] and CADeT [18] can be used for test case derivation because our approach results in complete products which need to be tested. Further model-based testing approaches are summarized in [19]

The scope of this contribution is marked with a dashed line: We aim to generate a minimal set of products covering all pairs of feature combinations. Therefore, testing this subset of products is equivalent to pairwise testing the whole SPL. To generate a subset of products the FM is first flattened followed by the subset extraction. The resulting products are subsets of the FM and can be traced to the test model. Only the subset of the test model which remains when tracing the FM subset to the test model is relevant for test case derivation for the corresponding product.

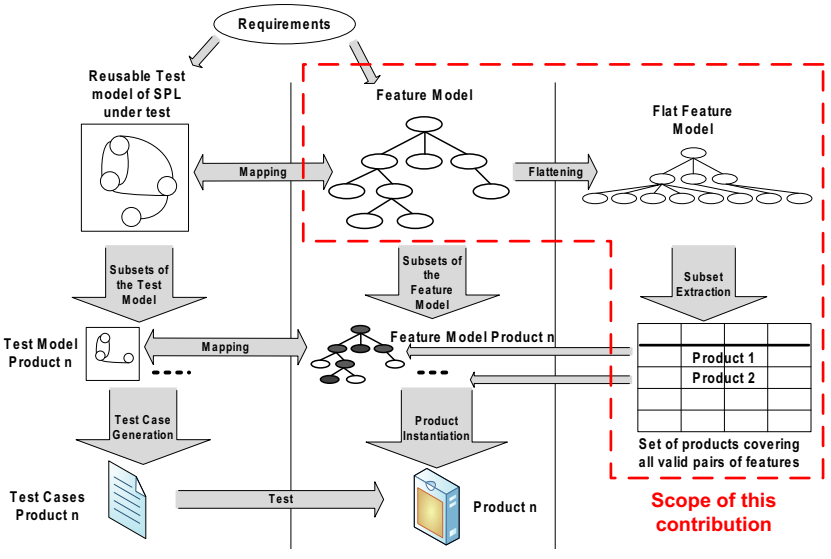


Fig. 2. Brief overview of the MoSo-PoLiTe testing process

3.1 Translation into Constraint Solving Problem

To apply pairwise testing to FMs we either have to adapt an existing pairwise algorithm so that it can handle the hierarchical structure, the different node notations, and constraints of the FM or we have to change the structure of the FM so that it can be processed using existing pairwise algorithms. We combine both ideas: First, the structure of the FM is changed so that it is processable by pairwise algorithms. Then, we implemented a pairwise algorithm based on IPO [13] and AETG [12]. Standard pairwise algorithms do not take constraints into account when certain pairs of features are combined in one product. As a consequence these algorithms must be extended by standard constraint solving techniques such as forward checking [20] which is well-suited for handling binary constraints between variables. To translate an FM into an ordinary CSP we implemented a flattening algorithm extracting parameters and parameter values of an FM. A subset extraction algorithm then implements pairwise combination and applies forward checking solving this CSP to generate a set of products containing all valid pairs of features.

We are aware that SAT-Solving approaches can be used to implement pairwise testing to FMs [21] as well. However, we intent to utilize pairwise testing in combination with our FMT approach described in [22,23] which integrates classification trees with FMs. There, we deal with (binary) constraints over larger parameter value domains. Especially, for such problems CSPs seem to be the natural choice [24,25].

3.2 Flattening

To extract parameters with corresponding values, a so-called flattening algorithm was developed reducing the depth of the FM. Furthermore, this flat FM serves as a temporary model in which every kind of FM can be translated by adapting the transformation rules. The algorithm consists of the following two steps:

1. Every feature with its associated notation and dependencies is iteratively pulled up until it is placed directly beneath the root node. Every feature then serves as a parameter.
2. The algorithm assigns every parameter its correspondent parameter value.

Several model transformation rules control the flattening process. We implemented these rules with MOFLON/Fujaba in the form of so-called SDM diagrams—a mixture of UML activity diagrams and graph transformation [26] as well as with Java in form of a pure::variants plug-in [27]. Each rule is iteratively applied to a subtree of an FM. A subtree always consists of three levels: the grandparent node, the parent node, and the child node. Different rules are required for the flattening process depending on the notations of the involved features. Our FM supports four different node notations and for every possible combination of parent and child notation a separate transformation rule is required. Therefore, $4 \times 4 = 16$ rules are needed. In the following we depict one rule to exemplarily describe our flattening approach. For a complete description of all rules refer to [28]. Fig. 3 depicts a transformation pulling up an *alternative*-group of child nodes with a parent node placed in an *or*-group. The parent *or*-group stays unchanged and the *alternative*-group is pulled up aside the parent. Due to the fact that the features **3MP** and **8MP** can only be chosen if **Camera** is selected, we have to add require dependencies. Furthermore, an additional feature is added into the alternative group: the ¬**Camera** feature which is required for the situation that **Camera** is not selected. Without adding this feature either **3MP** or **8MP** are always selected and, therefore, **Camera** is always required. Selecting ¬**Camera**, the feature **Camera** is excluded and we preserve the semantical equivalence between both FMs.

After the first step of the flattening algorithm, all features are placed directly beneath the root node serving as parameters. In the next step, we extract the corresponding values. Again different rules are applied to extract the values of the features.

– **optional:** An optional feature is changed to a mandatory feature with two child nodes. The optional feature **MMS** turns into a mandatory node with an

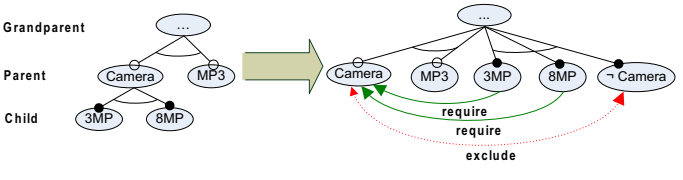


Fig. 3. Transformation rule pulling up an alternative-child with an or-parent

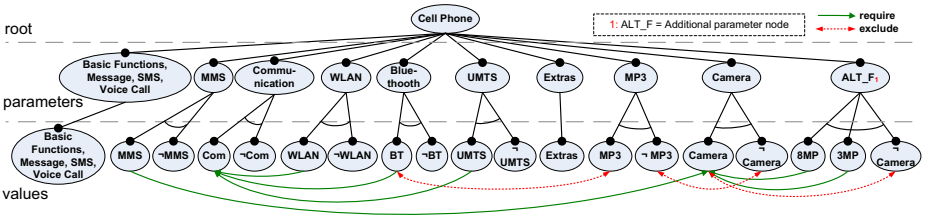


Fig. 4. Flat feature model with parameters and values

alternative child-group containing a feature **MMS** and ¬**MMS**. For product instantiation the feature **MMS** is selected and one element of the alternative group has to be chosen as well. Therefore, either the feature **MMS** or the feature ¬**MMS** is selected.

- **mandatory:** Mandatory nodes stay mandatory and obtain an additional child node with the same notation and name. (e.g. **Extras**)
- **or:** Extracting the parameter values of an **or**-group is the most complex rule. Each feature of the **or**-group is handled like an optional feature. To ensure that a least one element of the **or**-group has to be chosen within a product, the values for not including the features within a product exclude each other.
- **alternative:** An alternative group stays unchanged but we add a single placeholder feature in between the alternative group and the root node representing the parameter (**ALT_F**).

Fig. 4 shows the flat FM of our running example. This step ensures that every possible feature configuration is considered for the pairwise combination. The flattening process results in additional require and exclude dependencies between features. Additionally, we also have to consider existing cross-tree dependencies within the FM to ensure the semantical equivalence between the original and the flat FM. All require and exclude dependencies are transferred to the values of the features.

3.3 Subset Extraction

The Subset Extraction algorithm generates all valid pairwise combinations similar to IPO and AETG regarding cross-tree-dependencies. A valid pair is a combination of features not violating cross-tree dependencies, the hierarchical structure, and the different feature notations in the FM. Then, the algorithm incrementally combines those pairs of features to create valid products. The algorithm starts with the first pair and iteratively adds pairs of the remaining parameters. For each step forward checking [20] is applied to determine whether the selected pair can be combined with remaining pairs of parameters to create a valid product. If a certain pair results in such a deadlock, another pair

	B,M,S,V	Com	Extras	MMS	WLAN	BT	UMTS	MP3	Camera	ALF_F
P 1	B,M,S,V	Com	Extras	MMS	WLAN	BT	UMTS	¬MP3	Camera	8MP
P 2	B,M,S,V	¬Com	Extras	¬MMS	¬WLAN	¬BT	¬UMTS	MP3	Camera	8MP
P 3	B,M,S,V	Com	Extras	¬MMS	¬WLAN	BT	¬UMTS	¬MP3	Camera	3MP
P 4	B,M,S,V	¬Com	Extras	MMS	¬WLAN	¬BT	¬UMTS	¬MP3	Camera	3MP
P 5	B,M,S,V	Com	Extras	¬MMS	WLAN	¬BT	UMTS	MP3	Camera	3MP
P 6	B,M,S,V	Com	Extras	¬MMS	¬WLAN	¬BT	UMTS	MP3	¬Camera	¬Camera
P 7	B,M,S,V	¬Com	Extras	¬MMS	¬WLAN	¬BT	¬UMTS	MP3	¬Camera	¬Camera
P 8	B,M,S,V	Com	Extras	MMS	WLAN	¬BT	¬UMTS	MP3	Camera	8MP
P 9	B,M,S,V	Com	Extras	¬MMS	WLAN	¬BT	¬UMTS	MP3	¬Camera	¬Camera

Fig. 5. The resulting products covering all valid pairs of features of the running example

is selected instead. The algorithm continues until all pairwise combinations are covered by at least one product and will return the list of selected products. We applied our algorithm to the presented running example. The algorithm identified 9 products which are listed in Fig. 5 covering all pairwise interactions of features. Furthermore, the subset extractor can handle pre-selected products. To realize this functionality, the pairs of the pre-selected products are extracted and stored. When generating the set of pairs to cover these pairs are marked as already covered and the algorithm uses the remaining pairs. We define this functionality as "incremental pairwise combination".

Our pure::variants plugin realizes flattening as well as the subset extraction algorithm. First the FM is read out and stored in an own data structure. This data structure can be processed in pure::variants as well. On the basis of this data structure, flattening as well as the subset extractor are applied to the FM. The pre-selection functionality is used within pure::variants to give the user the opportunity to involve already existing/tested products. The resulting set of products then contains the pre-selected products along with further products to cover all valid pairs of features.

4 Testing the Implementation

To systematically test the implementation of the previously two-stage algorithm in pure::variants we need to check the following characteristics:

- Semantical equivalence: All transformation rules of the 1st phase preserve the semantics of the manipulated FMs.
- Consistency: we only generate valid products.
- Completeness: the generated products cover all pairs of features that do not exclude each other.
- Efficiency: our heuristics generate a small set of products. We list some results with the corresponding runtime of our algorithm.

We used some of the FMs listed in [8] to test our implementation. Furthermore, we implemented a feature model generator similar to SPLOT [8] generating feature models in pure::variants.

4.1 Systematic Validation

Our pure::variants feature model generator (pv-FMG) creates random FMs considering certain input parameters. A root node is created which obtains a random number of children restricted by an adjustable maximum. The FMG sets the notations of the features beneath this root node. When generating an FM two possible configurations are possible: Every node receives a random notation or the distribution of mandatory, optional, or, and alternative features can be configured. The percentage of every notation, "mandatory", "alternative", "or" and, "optional" can be adjusted by parameters. For our test runs the pv-FMG uses the following distribution of notations: Alternative (21,7%), Or (24,1%), Optional (23,0%), and Mandatory (28,1 %) according to [8]. Every child is handled as a root node and obtains its own children in an analogous manner. Thus, the algorithm creates a complete FM iteratively, aborted by a defined maximum depth. To generate asymmetric FMs, the random number of added children can be zero, too. In order to approximate real SPL FMs, the generator also creates constraints. To satisfy this functionality, a valid random pair of features is selected and a constraint (either require or exclude) is set. The number of inserted constraints depends on the total number of nodes and a parameter configures the percentage of nodes involved within a constraint.

Semantical Equivalence
Two FMs are semantically equivalent if they describe the same set of products with respect to a given set of features. We validate the flattening algorithm by comparing the propositional formulas of the original and the flat FM [29]. With regard to our running example, the following logical expression is computed for both FMs:

$$\begin{aligned}
&CellPhone \wedge BasicFunctions \wedge Message \wedge VoiceCall \wedge SMS \wedge \\
&(\neg WLAN \vee Communication) \wedge (\neg Bluetooth \vee Communication) \wedge \\
&(\neg UMTS \vee Communication) \wedge Extras \wedge (MP3 \vee Camera) \wedge \\
&(\neg Camera \vee (3MP \oplus 8MP)) \wedge (\neg 3MP \vee Camera) \wedge \\
&(\neg 8MP \vee Camera) \wedge (\neg MMS \vee Camera) \wedge \neg(Bluetooth \wedge MP3)
\end{aligned} \tag{1}$$

We used the FMG to create additional FMs and apply the flattening algorithm. Furthermore, for each model transformation we have proved beforehand by hand that it always preserves the semantics of manipulated FM. For all FMs the pure::variants implementation preserves the semantic equivalence.

Consistency
The consistency can be checked automatically within pure::variants. Pure::variants has an integrated solver to check for inconsistency and whether a feature configuration is a valid product or not. We rely on the fact that the internal solver works correctly.

Completeness
To further evaluate our subset derivation algorithm we need to examine whether we cover all valid pairs of features with our set of products. This cannot be

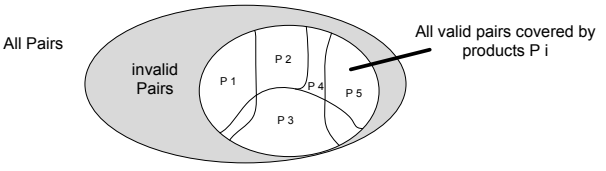


Fig. 6. Valid and invalid pairs of features

checked using pure::variants itself. The implementation of the subset derivation algorithm is complete if all products that contain an uncovered feature pair are inconsistent. Fig. 6 depicts this assumption.

We apply a SAT-Solver to prove that all uncovered pairs cannot be part of a valid product in order to validate the implementation of our algorithm.

1. we generate all pairs of features ignoring require and exclude dependencies within the FM and write them into a list
2. we remove all pairs which are covered by our products from the list
3. we check if all remaining pairs are invalid

For our experiments we use MiniSAT as SAT solver and solve the following equation:

$$featuremodel \wedge (invalid\_pair_1 \vee invalid\_pair_2 \vee ... \vee invalid\_pair_n) \qquad (2)$$

If equation (2) is unsatisfiable, all pairs must have been invalid with respect to the FM. Therefore, the products that are generated by our algorithm, must cover all possible pairs of features. We applied MiniSAT to the FM in Fig. 1 combined with all pairs that are not part of any generated product instance and, therefore, must be invalid. We found 129 potentially invalid pairs and reviewed them with one run of MiniSAT. MiniSAT took less than 1 second and 2 MB of memory to find that the expression is unsatisfiable. This proves that all pairs not covered by any of the generated product instances are invalid pairs like shown in Fig. 6.

Evaluation of Efficiency
To test the efficiency of our algorithm, we applied it to some FMs listed in [8]. The results are listed in Table 1. The last two columns contain the number of products realizing pairwise coverage and the runtime of the algorithm in milliseconds on a 2Ghz Single Core machine with 2 GB RAM. For further evaluation we will compare our approach with the one described in [21] in our future work. The next subsection describes some statistics when applying our approach to random FMs.

4.2 Statistics

To test our approach, we generate a set of 1023 random FMs automatically. The probability of the node notations is selected with: Alternative (21,7%), Or

Table 1. Further Examples

Feature Model	Features	Possible Products	Pairwise Products	Runtime [ms]
Smart Home	35	1,048,576	11	0
Inventory	37	2,028,096	12	16
Sienna	35	2,520	24	16
Web Portal	38	2,120,800	26	15
Doc_Generation	44	$5.57 \cdot 10^{7}$	18	0
Arcade Game	61	$3.3 \cdot 10^{9}$	25	32
Model_Transformation	88	$1.65 \cdot 10^{13}$	40	46
Coche ecologico	94	$2.32 \cdot 10^{7}$	114	47
Electronic Shopping	287	$2.26 \cdot 10^{49}$	62	797

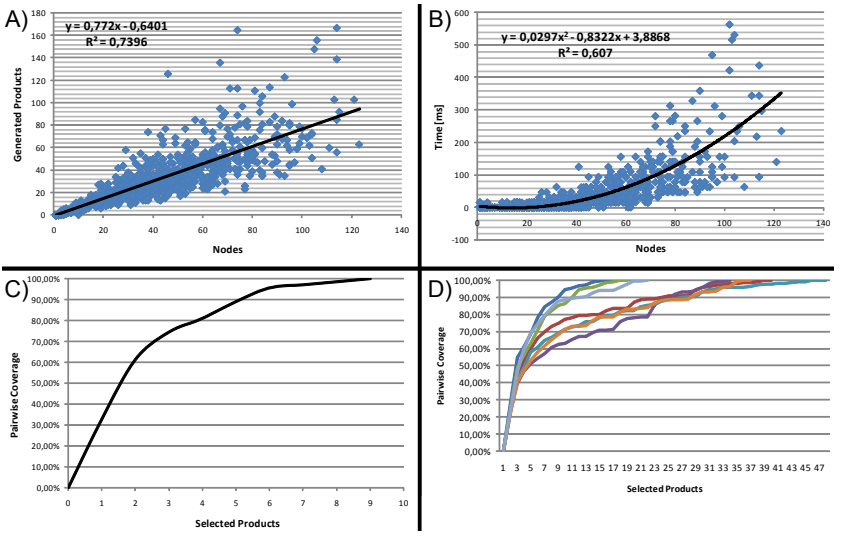


Fig. 7. Statistics

(24,1%), Optional (23,0%), and Mandatory (28,1 %) according to [8]. We set the maximum depth to 5 and the maximal number of children per node to 4. Therefore, the maximum possible number of features is 256. The generated 1023 feature models have a mean number of 35 features with a standard deviation of 28 features. Fig. 7 A) shows the relation between the number of nodes in the FM and the number of generated products. We apply linear regression to the set of values and found a slope of `0.772` and a y-intercept of `-0.6401`. The square of the correlation coefficient is `0.7396` which shows a strong linear dependence of the two values. Therefore, the average of the number of generated products will increase by 77.2% when the number of nodes in the FM is doubled. The relation between the number of nodes in the FM, the calculation time and the corresponding function is depicted in Fig. 7 B). In Fig. 7 C) the number of

product instances is related to the number of pairs covered. The left part shows the results for the FM presented in Fig.1. The coverage increases fast until 6 products are generated and slowly saturates to 100%, afterwards. Fig. 7 D) shows the results for seven automatically generated FMs.

5 Related Work

In this section we summarize related work with regard to SPL-Testing and a flattening algorithm which is similar to our approach.

5.1 SPL-Testing

Studying related work focusing on SPL-Testing we identified three best practices:

Contra-SPL-philosophy: These approaches contradict with the SPL-reuse-philosophies. Every product derived from the SPL is tested individually ignoring reuse as well as the division into domain- and application engineering. In [30] the authors refer to this approach as product-by-product testing. However, considering the number of derivable products of today's SPLs this approach is not feasible any more.

Reuse-Techniques: Methods of this category utilize reuse-techniques to reduce the test effort. These approaches either make use of regression testing techniques to incrementally test products or realize the reuse of domain tests during application testing. Reusing domain tests created during domain engineering for product tests is a very popular approach especially in the model-based testing community. A summary of model based testing approaches for SPLs can be found in [19]. Although all approaches utilizing reuse techniques benefit from reducing the effort for testing, each product has to be tested individually. The next category focuses on identifying a subset of products to approximate a complete SPL-test according to some criteria.

Subset-Heuristics: This approach aims at reducing the effort for testing by extracting a subset of feature combinations or products. Instead of testing every product of the SPL, a subset for testing is created. We identified two different methodologies: Methods generating a subset of products which are representative for testing purposes for the whole SPL and approaches using combinatorial testing. In [31] the author introduces an approach generating a representative set for each requirement. The major disadvantage of this approach is the fact that it does not scale with real-world SPLs and that the effort to set up the representative set is enormous.

McGregor initially introduced combinatorial testing to SPLs in [32]. However, he neither describes how combinatorial testing may be applied to SPLs nor how SPL-models like FMs or OVMs can be mapped onto an appropriate representation to apply existing combinatorial testing algorithms. Cohen et al. use the OVM approach to model the variable and common parts of the SPL which are mapped onto a relational model. This relational model serves as a semantic basis

for defining coverage criteria for the SPL under test [33]. Furthermore, Cohen et al. describe the development of combinatorial interaction testing (CIT) achieving a desired level of coverage. In [10] the authors use the CIT approach to systematically select products that should be tested. The approach described in [21] is similar to the Cohen et al. approach. The significant difference is the fact that Perrouin et al. utilize SAT-solvers and do not use a relational model. Furthermore, this approach focuses on the scalability. Nevertheless, the output of the presented algorithm is not deterministic due to the random components. Therefore, the number of found products may vary strongly. However, we combine graph transformation, a well-known pairwise algorithm together with forward checking to generate a set of products achieving 100% pairwise interaction coverage in the whole SPL on the basis of the corresponding FM. The reason for choosing a CSP approach for pairwise testing is that we want to apply this approach to the FMT approach which utilizes large ranges of values. Especially, for such problems a CSP-based approach seems to be a natural choice [24,25].

5.2 Cartesian Flattening

In [34] the authors introduce a Cartesian Flattening of FMs which is similar to our flattening algorithm. There, the motivation is to translate the FM into a knapsack problem which is then used to generate highly optimal architectural variants/products of the SPL. There are some significant differences to our flattening approach concerning the handling of cardinality groups (or-groups in our approach) which are translated into an XOR-group (alternative-group in our approach) with a maximum number boundary in [34]. For testing purposes all valid feature combinations need to be identified and we would lose the semantical equivalence between the original FM and the flat FM if we would use a boundary, limiting the maximum number of combinations. Due to the different field of application, White et al. apply different transformation rules to prepare the FM for their algorithms. However, this approach offers an additional evidence that it is possible to change the structure of the FM in order to be able afterwards to apply well known algorithms for different purposes.

6 Conclusion and Future Work

In this paper we introduced a description of how to apply pairwise testing to feature models. The pairwise testing approach is based on the assumption that the SPL under test is rather modular in its structure and that the majority of possible errors occur when two modules/features interact. The introduced pairwise algorithm can also be applied apart from SPL engineering. According to the best of our knowledge, this is the first pairwise algorithm combining binary constraint solving, forward checking, and a pre-selection functionality. Furthermore, we introduced our current model-based testing approach for SPLs in which we integrated the pairwise testing approach. Please note that our MoSo-PoLiTe approach can be applied with every model-based testing approach for SPLs offering test case derivation for configurable products. A summary of existing approaches can be found in Oster et al. [19].

In our future work we plan to apply our pairwise testing approach to a real SPL provided by an industrial partner. We are interested in the degree of coverage for real world examples to evaluate whether pairwise testing is sufficient. During these evaluations we will focus on data-flow and control-flow coverage of the underlying code. To complete our evaluation process, we will compare our approach to SAT-Solver methodologies and with random product selection testing. As a matter of fact, we can also apply three- and four-wise testing to SPLs using our flat feature model. In our future work, we will evaluate the degree of coverage of pairwise, three-, and four-wise testing. A technical report with a detailed description of our pairwise algorithm including pre-selection is under development.

References

1. Clements, P., Northrop, L.: Software product lines: practices and patterns. Addison-Wesley Longman Publishing Co., Inc., Boston (2001)
2. Pohl, K., Böckle, G., Van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, New York (2005)
3. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional, Reading (June 2000)
4. Bosch, J.: Design and Use of Software Architectures - Adopting and Evolving a Product Line Approach (2000)
5. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multilevel configuration of feature models. Software Process: Improvement and Practice 10(2), 143–169 (2005)
6. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute (November 1990)
7. Heymans, P., Schobbens, P.Y., Trigaux, J.C., Bontemps, Y., Matulevicius, R., Classen, A.: Evaluating formal properties of feature diagram languages. Software, IET 2(3), 281–302 (2008)
8. SPLOT-Research: http://www.splot-research.org
9. Beizer, B.: Software testing techniques, 2nd edn. Van Nostrand Reinhold Co., New York (1990)
10. Cohen, M., Dwyer, M., Shi, J.: Interaction testing of highly-configurable systems in the presence of constraints. In: ISSTA, pp. 129–139 (2007)
11. Stevens, B., Mendelsohn, E.: Efficient software testing protocols. In: Conference of the Centre for Advanced Studies on Collaborative Research. IBM Press (1998)
12. Cohen, D.M., Dalal, S.R., Kajla, A., Patton, G.: The automatic efficient tests generator. In: Fifth ISSRE IEEE, pp. 303–309 (1994)
13. Lei, Y., Tai, K.: In-parameter-order: a test generation strategy for pairwise testing. In: IEEE High Assurance Systems Engineering Symposium, pp. 254–261 (1998)
14. feasiPLE Consortium (2006-2009), http://www.feasiple.de
15. Wübbeke, A.: Towards an Efficient Reuse of Test Cases for Software Product Lines. In: Thiel, S., Pohl, K. (eds.) Proceedings of the 12th International Software Product Line Conference Second Volume, pp. 361–368 (2008)
16. Weißleder, S., Sokenou, D., Schlinglo, B.: Reusing State Machines for Automatic Test Generation in Product Lines. In: Proceedings of the 1st Workshop on Model-based Testing in Practice (MoTiP 2008) (2008)

17. Reuys, A., Kamsties, E., Pohl, K., Reis, S.: Model-based System Testing of Software Product Families. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 519–534. Springer, Heidelberg (2005)
18. Olimpiew, E.M.: Model-Based Testing for Software Product Lines. PhD thesis, George Mason University (2008)
19. Oster, S., Wübbeke, A., Engels, G., Schürr, A.: Model-Based Software Product Lines Testing Survey. In: Zander, J., Schieferdecker, I., Mosterman, P. (eds.) Model-based Testing for Embedded Systems. CRC Press/Taylor&Francis (to appear, 2010)
20. Haralick, R., Elliott, G.: Increasing tree search efficiency for constraint satisfaction problems. Artificial intelligence 14(3), 263–313 (1980)
21. Perrouin, G., Sen, S., Klein, J., Baudry, B., Traon, Y.L.: Automated and scalable t-wise test case generation strategies for software product lines. In: Third International Conference on Software Testing, Verification and Validation (2010)
22. Oster, S., Markert, F., Schürr, A.: Integrated Modeling of Software Product Lines with Feature Models and Classification Trees. In: Proceedings of the 13th International Software Product Line Conference (SPLC 2009). MAPLE 2009 Workshop Proceedings. Springer, Heidelberg (2009)
23. Schürr, A., Oster, S., Markert, F.: Model-Driven Software Product Line Testing: An Integrated Approach. In: 36th International Conference on Current Trends in Theory and Practice of Computer Science. LNCS, pp. 112–131. Springer, Heidelberg (2009)
24. Bennaceur, H.: A Comparison between SAT and CSP Techniques. Constraints 9(2), 123–138 (2004)
25. Westphal, M., Wölfl, S.: Qualitative csp, finite csp, and sat: comparing methods for qualitative constraint-based reasoning. In: IJCAI 2009: Proceedings of the 21st international jont conference on Artifical intelligence, pp. 628–633. Morgan Kaufmann Publishers Inc., San Francisco (2009)
26. Oster, S., Schürr, A., Weisemöller, I.: Towards Software Product Line Testing using Story Driven Modelling. In: Aßmann, U., Johannes, J., Zündorf, A. (eds.) Proceedings of the 6th Int. Fujaba Days, TU Dresden, pp. 48–51 (2008)
27. Oster, S., Ritter, P., Schürr, A.: Featuremodellbasiertes und kombinatorisches Testen von Software-Produktlinien. In: Proceedings of the SE 2010. GI-Edition Lecture Notes in Informatics. Gesellschaft für Informatik (2010)
28. MoSo-PoLiTe: http://www.sharq.tu-darmstadt.de/projects/mosopolite/
29. Thum, T., Batory, D., Kastner, C.: Reasoning about edits to feature models. In: ICSE 2009: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, pp. 254–264. IEEE Computer Society, Washington (2009)
30. Tevanlinna, A., Taina, J., Kauppinen, R.: Product family testing: a survey. ACM SIGSOFT Software Engineering Notes 29, 12 (2004)
31. Scheidemann, K.: Verifying families of system configurations. Doctoral Thesis TU Munich (2007)
32. McGregor, J.D.: Testing a software product line. Technical Report CMU/SEI-2001-TR-022 (2001)
33. Cohen, M.B., Dwyer, M.B., Shi, J.: Coverage and adequacy in software product line testing. In: ROSATEA 2006: Proceedings of the ISSTA 2006 workshop, pp. 53–63. ACM, New York (2006)
34. White, J., Dougherty, B., Schmidt, D.C.: Selecting highly optimal architectural feature sets with filtered cartesian flattening. Journal of Systems and Software 82(8), 1268–1284 (2009)

Linking Feature Models to Code Artifacts Using Executable Acceptance Tests*

Yaser Ghanam and Frank Maurer

Department of Computer Science
2500 University Dr. NW, Calgary
Alberta, Canada T2N 1N4
{yghanam,fmaurer}@ucalgary.ca

Abstract. A feature model is a representation of the requirements in a given system abstracted at the feature level. Linking conceptual requirements in feature models to actual implementation artifacts provides for many advantages such as increased program comprehension, implementation completeness assessment, impact analysis, and reuse opportunities. However, in practice, as systems evolve, traceability links between the model and the code artifacts may become broken or outdated. In this paper, we contribute an approach to provide traceability links in a way that ensures consistency between the feature model and the code artifacts, enables the evolution of variability in the feature model, and supports the product derivation process. We do that by using executable acceptance tests as a direct traceability link between feature models and code artifacts. We evaluate our approach and present a brief overview of the tool support we provide.

Keywords: agile product line engineering, feature models, traceability, variability evolution, executable acceptance tests.

1 Introduction

Feature modelling has become an essential aspect of software engineering in general and software product line engineering (SPLE) in particular. A feature model is a representation of the requirements in a given system abstracted at the feature level [30]. A feature can be broadly defined as a chunk of functionality that delivers value to the end user. In SPLE, feature models represent a hierarchy of features and sub-features in a product line and include information about variability in the product line and constraints of feature selection.

Linking conceptual requirements in feature models to actual implementation artifacts provides for advantages such as increased program comprehension, implementation completeness assessment, impact analysis, and reuse opportunities [2]. Nevertheless, traceability is a non-trivial problem. Berg et al. [3] analyzed traceability between the problem space (i.e. the model) and the solution space (i.e. the development artifacts) in a software product line context. The results suggested that the feature model provided an

* This research is supported by iCore – Alberta Innovates Technology Futures.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 211–225, 2010.

excellent visualization means at individual levels of abstraction. However, it did not improve the traceability between artifacts across development spaces. Furthermore, in practice, as the product line evolves, traceability relationships between the model and the code artifacts may become broken or outdated [29]. This happens either because changes in the model are not completely and consistently realized in the code artifacts; or because changes due to continuous development and maintenance of the code artifacts are not reflected back in the model. This problem is not unique to SPLE. In fact, outdated traceability between requirement specifications and other development artifacts has always been an issue in software engineering [13, 6].

Traceability links provided by some commercial tools (e.g. DOORS [7]) mitigate this issue, but leave some other problems unsolved. For example, say feature A and feature B are independent features in the product line. During the maintenance of feature A, the developer introduced a change that unintentionally caused a technical conflict between feature A and feature B. Although the tool will maintain the traceability links between each piece of code and the correspondent feature, it cannot, uncover the newly introduced conflict in order to reflect it back in the model.

In this paper, we propose the use of executable acceptance tests as a direct traceability link between feature models and code artifacts. In the next subsection, we give an overview of executable acceptance tests and their characteristics.

1.1 Executable Acceptance Tests

Requirement specifications – in its traditional format – exist in a number of documents and are written in a natural language. The correctness of the behaviour of a system is determined against these specifications using test cases or scenarios. On the other hand, executable specifications are written in a semi-formal language that aims to reduce ambiguities and inconsistencies. Executable specifications take various formats ranging from very formal [11] to English-like [22]. The English-like ones are often called scenario tests [17], story tests [19], or acceptance tests [26]. They are usually used in organizations where Agile Software Development [20] is practiced. These names highlight the role of these artifacts as:

1. Cohesive documentation of the specifications of a given feature.
2. Accurate, high-level validity tests: by being executable, these specifications can be run (executed) against the system directly in order to test the correctness of its behaviour.

Throughout this paper, we will use the general term executable acceptance test (EAT) to refer to the English-like specifications that can play the two roles above. In this paper, we present an idea on how EATs can be used as a traceability link between feature models and code artifacts. Fig. 1 shows an example of an EAT. If the behaviour of the system matches the expected one as specified in the EAT, the test passes. Otherwise, the test fails indicating either a technical problem in the code, or a business problem in understanding the specifications of the system. To link the EAT to actual production code, a thin layer of test code – called fixture – is used. EATs are usually executed using tools like FIT [10] and GreenPepper [14].

Home owner is notified after two failed attempts				
Start	Screen.Login			
Enter	Name	John	PIN	1234
Check	Info is valid	False		
Enter	Name	John	PIN	4321
Check	Info is valid	False		
Check	Owner is notified			

Fig. 1. Example of an EAT

1.2 Traceability from EATs to Code Artifacts

The fundamental basis of our approach is that EATs natively provide the necessary links to code artifacts. The reason why acceptance tests can be executed against the system is that they are linked to a thin layer of test code, and from there to actual production code. Fig. 2 shows an example of this traceability. At the first layer, only one row of a row-fixture EAT is shown for simplicity. This row is linked – by a test automation framework (e.g. FIT) – to a method in the test code called *addResidentWithPIN(…)*. This method in turns uses the *addResident(…)* method in the production code, specifically in the *HomeResidentsList* class. When the test is executed, an attempt to add a resident with the given parameters will be made. In this scenario, if the attempt is not successful – for a variety of reasons such as the PIN being too short or too long – the EAT will fail. Otherwise, it will pass. Usually, a suite of EATs is executed rather than a single EAT. Moreover, with appropriate test coverage, tools generate reports stating which methods where involved in the execution process of a certain EAT. Later in the paper, we will discuss how this traceability is useful in linking features models with the code artifacts.

The rest of this paper is structured as follows. Section 2 is a review of relevant literature. Section 3 presents the proposed approach. Section 4 elaborates on the positive implications of the approach. Section 5 is an evaluation of our approach in comparison to other traditional approaches. Finally, we conclude in Section 6.

2 Literature Review

There is a large body of research on feature modeling in software engineering in general, and SPLE in particular. FODA [18] was one of the earliest techniques off which many other techniques were based (e.g. [16] and [8]). In our work, we use feature trees as described in traditional modeling techniques such as FODA, but the generality of our work is not affected by that choice.

Efforts to study traceability links between feature models and other development artifacts include the one by Filho et al. [15] in which they proposed the integration of feature models with the UML meta-model to facilitate the instantiation process. Another effort was the one by Ramesh et al. [28] in which use cases (representing requirements) were linked to design artifacts and from there to code artifacts. To group requirements at a more meaningful and comprehendible level of abstraction, Riebisch [29] suggested the use of feature models as an intermediate element between use cases and other artifacts. The main issue with this approach is that in real settings a massive effort is required to establish and maintain the traceability links due to the informal descriptions of the requirements – which made automation impossible [25].

To solve the language informality issue, new techniques were proposed. For example, Antoniol et al. [2] proposed an information retrieval method to link flat requirements to code artifacts. The caveat of the approach is that it is based on the hypothesis that programmers use names for program items (e.g. classes, methods, variables) that are also found in the text documents. There is also the issue of managing and maintaining the established traceability links. In a panel report, Huang [15] discusses the state-of-the-practice in traceability techniques. The report asserts that requirement trace matrices (RTMs) are often maintained either manually or using a management tool; and the amount of effort needed to keep these links up-to-date is enormous. Commercial tools are available to support traceability. CaliberRM [4], DOORS [7] and other tools are used to manage and visualize traceability links. However, these links have to be established manually, and the tools do not address issues specific to feature models such as variability in requirement. Some software product line tools like pure::variants [27] provide add-ins to allow requirement models in traditional management tools to be remodeled as feature models.

Our contribution in this paper is novel because we link feature models to specifications that are executable. We also show in the sections to follow how this linkage provides advantages specific to feature models and software product lines.

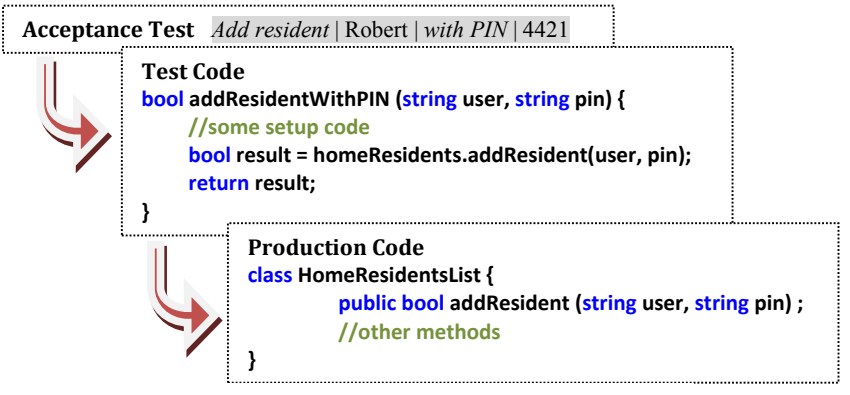


Fig. 2. Traceability through EATs

3 Using Feature Models with EATs

We propose extending feature models by including EATs as concrete descriptors of features at the lowest level of the feature tree. EATs should be associated with features that originally would be considered leaf nodes in the tree as shown in Fig. 3. For instance, the feature "Access by PIN" is associated with three EATs. These EATs describe scenarios that need to be satisfied in the implementation of this specific feature. [1]

[1] This is a simple example of a feature model. All features are mandatory unless there is a white circle indicating their optionality. For instance, the "Access Control" feature is optional. Grouping features (or sub-features) with an arch indicates that these features are alternatives. That is, only one feature can be selected from the group. If more than one feature can be selected from a group, a multiplicity constraint of the form [min..max] will be included.

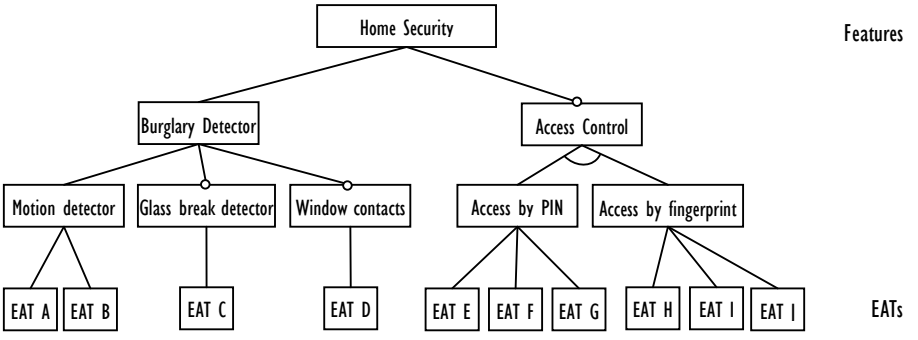


Fig. 3. The proposed extension to feature models

Linking between an EAT node in the model and the actual specification happens by associating a test unit to the EAT node. An EAT node can link to a test table, a test page, or a test suite. We intentionally do not put any constraints on the granularity of the test unit to leave it flexible for various contexts. Nevertheless, a single test table may be insufficient given that usually more than one table is needed to specify some behaviour. This makes a single table less cohesive than desired. On the other hand, a test suite may be too large because it involves more than one feature creating dependencies between test units. Therefore, we suggest the use of a test page as a usual test unit that provides reasonable cohesion and independence. Depending on the testing tools, test pages can take various formats such as html files or excel sheets.

3.1 Linking Features to EATs

Following the earlier definition of a feature as a chunk of functionality that delivers value to the end user, one EAT generally is not sufficient to represent a feature in a system. In practice, a group of EATs represent the different scenarios or stories expected in a given feature in a system. This implies that in order to somehow link features in a feature model to EATs, one-to-one relationships are not practical. Rather, each feature in the feature model should be linked to one or more EATs (Fig. 4). The "Access by PIN" feature is specified using three EATs. In order for the behaviour of this feature to be deemed correct, all three EATs should pass. Moreover, in some cases, a single EAT can be at a level high enough to cut across a number of features in the system. Consider, for example, a high-level EAT such as "Owner entering premises" as in Fig. 4. Say in order for the scenario specified in this EAT to pass, more than one feature should be involved (i.e. EAT X cuts across a number of features). This implies that a many-to-many relationship is needed in order to accurately represent the relationship between EATs and features in a feature model.

Linking features to EATs has consequences. For one, the selection of a feature in the product derivation phase automatically implies the inclusion of all its EATs. Secondly, EATs shall inherit all the dependencies and constraints originally imposed on their parent nodes. For example, according to the model in Fig. 4, the two features "Access by PIN" and "Access by fingerprint" are mutually exclusive. This implies that the groups: {EAT E, EAT F, EAT G} and {EAT H, EAT I, EAT J} are mutually

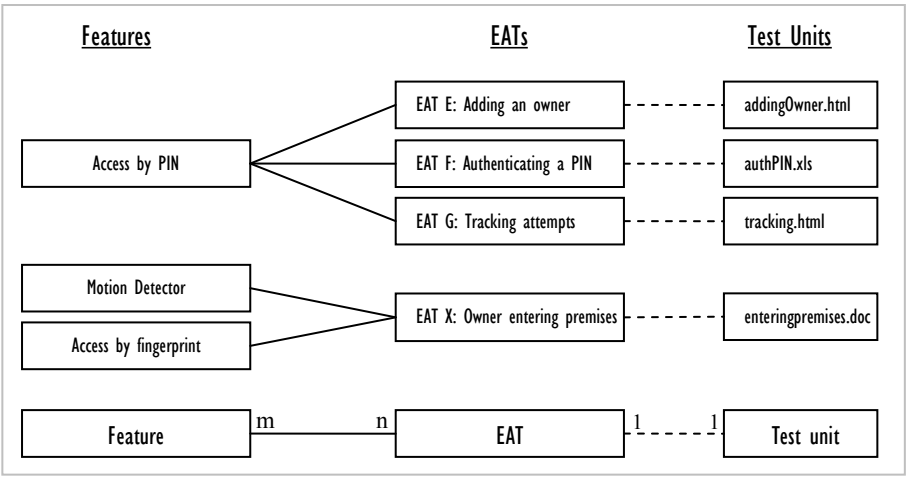


Fig. 4. Relationships between features, EATs, and test units

exclusive too. The importance of explicating these consequences will be discussed later in the paper.

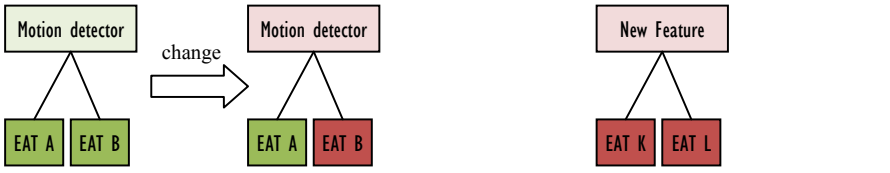
4 Implications of Using EATs as Traceability Links

In the previous sections, we discussed how features in the feature model can be linked to EATs in order to provide traceability links between the feature model and the code artifacts. This section analyzes the implications of using EATs by highlighting three main ways through which EATs provide significant contribution to feature models.

4.1 Consistency between the Feature Model and the Code Artifacts

EATs provide a means to ensure that the problem space (i.e. the specifications), and the solution space (i.e. the implementation) are consistent. This consistency is due to the fact that these specifications can be executed against the implementation, and the result of their execution gives an unambiguous insight of whether or not the intended requirements currently exist in the system. In our approach, we provide a link between feature models and EATs in order to inherit this important property. Within this context, we realize two key advantages of our approach:

Continuous Two-way Feedback. Maintaining a practice where every feature in the feature model has to be associated with some EATs is valuable. Changes due to continuous development and maintenance of the code artifacts are reflected back in the model, because – at any point of time – the EATs are either in a passing state (visualized as green) or a failing state (visualized as red). For instance, Fig. 5 shows how a change in the code (e.g. bug fix) caused EAT B to fail – also causing the "Motion Detector" to be denoted as incomplete. The opposite direction of feedback occurs when introducing a new feature to the model. The accompanied EATs will initially be in a failing state indicating that the feature is not implemented yet.



(a) a change in the code caused an EAT to fail providing immediate feedback in the feature model

(b) when adding a feature to the model, initially EATs fail indicating incomplete implementation

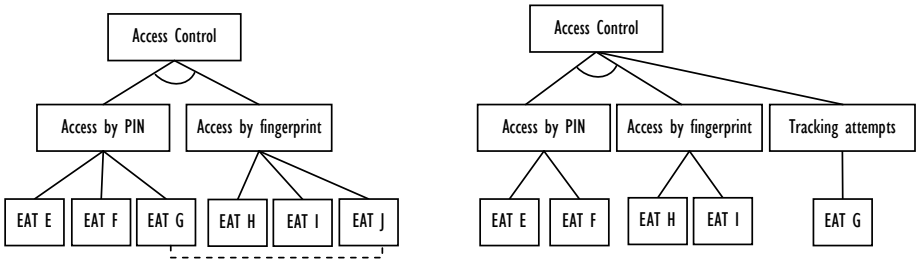
Fig. 5. Continuous two-way feedback

Exploiting Hidden Variability Concerns. Using EATs helps in revealing unwanted feature interactions that otherwise might be hidden. It also supports the realization of common aspects of features. We illustrate these points further by going through a number of scenarios.

Scenario 1: In some cases, the same EAT can be used as part of the specifications of two different features. If the features are originally mutually exclusive, and the same EAT passes in both, then this EAT is agnostic to the source of variation in the features. This means that the specifications in this EAT are part of the common portion of the parent node, which exploits a commonality aspect that was not originally apparent. Fig. 6 shows that because EAT G and EAT J are the same (we use a dashed line to denote this – it is also possible to give them the same name), we can abstract the commonality as a mandatory sub-feature under "Access Control".

Scenario 2: Using EATs allow finding unwanted feature interactions. EATs for independent features may pass when the features are selected separately; but fail when selected together. This is indicative of an unwanted feature interaction. This conflict is either a problem in the implementation and should be resolved, or an unavoidable real conflict that should then be reflected in the model as an "excludes" dependency or using a multiplicity constraint.

Scenario 3: Some EATs for independent features fail when these features are selected separately, but when selected together, they pass. This is indicative of a dependency



(a) EAT G and EAT J are the same, and they both pass in mutually exclusive features.

(b) exploiting the commonality aspect as a mandatory sub-feature.

Fig. 6. Abstracting the commonality as a mandatory sub-feature

between the features. It can be either due to unnecessary coupling in the implementation itself that should be resolved, or due to a necessary "requires" dependency that should then be reflected in the model.

4.2 Supporting the Evolution of Variability in the Extended Feature Model

Using EATs as a basis for evolving variability in the feature model is rewarding in a number of ways. Consider the following scenarios:

Scenario 1: A new feature or sub-feature is added to the feature model. In case the newly added feature causes EATs of other features that were originally passing to fail, this is a sign that a new conflict was introduced by the new feature. Without the direct feedback of failing tests, it is less likely for this conflict to be immediately exposed.

Scenario 2: An existing feature or sub-feature is removed from the feature model. If this feature was originally related to other features, then all dependencies are to be resolved before removing the feature safely. However, in case there was a hidden (unexploited) dependency between this feature and other features, removing this feature and its corresponding code might have a destructive effect on the other features. The fastest way to discover such effects is by looking for EATs that started to fail only after removing the feature.

Scenario 3: A new variant is to be added to a group of variants under a given feature. For developers, using EATs provides guidance on where and how this new variant should be accommodated in the system. For example, suppose we want to add a new alternative "Access by Magnet Card" under "Access Control". First of all, we may be able to reuse the EATs of the other sibling alternatives and tweak them to reflect the requirements of the new alternative. And because EATs are traceable to code artifacts, we can look at the implementation of the sibling alternatives in order to have a better comprehension on where in the code we should incorporate the new variant, and how it should be handled. With appropriate tool support, we can also automate the process of adding a variant by using the sibling nodes as templates, and directing the developer to the exact place in the code base where the new logic should be added [12]. This is particularly important for legacy systems with poor or outdated design documentation or for development environments where design documentation might not be available at all.

Scenario 4: Abstracting a variability aspect to the common layer. Say an EAT is used as part of the specifications of two mutually exclusive features, and this EAT passes in both. This means that the specifications in this EAT can be abstract to become part of the common layer of the parent node (as a mandatory sub-feature – this was discussed in the previous section).

4.3 Deriving Products Using the Extended Feature Model

In a software product line context, feature models are used to select features and variants that constitute a product instance. The selection process should take into consideration the constraints and dependencies between features and variants, as conveyed in the feature model. Nowadays, tool support is available to make this process easier, faster and less error-prone. Once the features and configurations have been selected,

an instance is derived that has the required feature composition and configuration. This section discusses the beneficial roles EATs can play in the product derivation process (aka. product instantiation process).

Selecting Configurations. During the derivation process, we usually need to set certain parameters (e.g. compiler directives, configuration classes) in order to select certain configurations for the product instance at hand. We can rely on EATs to automatically set up these parameters. This can be done because for an EAT to pass (independently of other EATs), it needs to set the correct parameter before it can execute the production code. When we finish the selection process of features in the feature model, we can run all the EATs that are relevant to the current selection. Given that all EATs have passed for the current selection, this means that all parameters in the system have been set properly, and the system is now ready to produce the right instance (Fig. 7). Another role of EATs in this context can be described as "*configuration by example*." That is, EATs provide a good starting point for the developers to learn how to configure a certain feature.

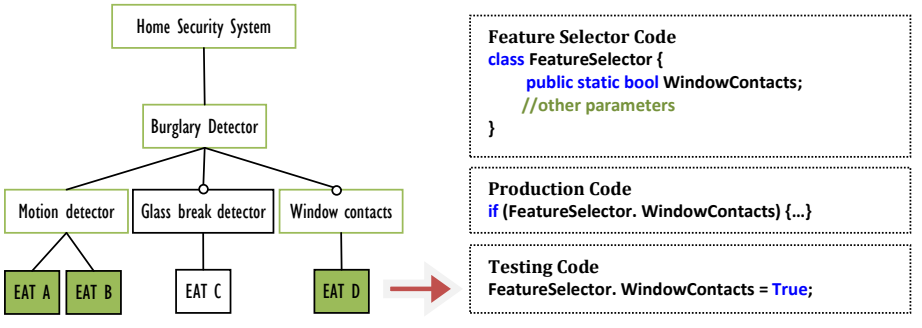


Fig. 7. Using EATs to select configurations

Extracting Required Artifacts. In some derivation techniques, a subset of code artifacts are extracted from a common base according to which features in the feature model were selected. EATs can play an important role in supporting this process. After the selection process of features in the feature model, we can run all the EATs that are relevant to the current selection as shown in Fig. 8 (CU refers to code unit and TU refers to test unit). Static code analysis can provide details on which code artifacts are needed to produce the desired instance by computing the transitive closure of all calls in the fixture classes used in the EATs of the instance.

4.4 Tool Support

In order to realize the benefits we discussed in the previous sub-sections, we built a tool that supports traceability links between the feature model and code artifacts via EATs[2]. To avoid reinventing the wheel, an open-source modeling tool was chosen in

[2] We thank Felix Riegger for his contribution in providing the tool support.

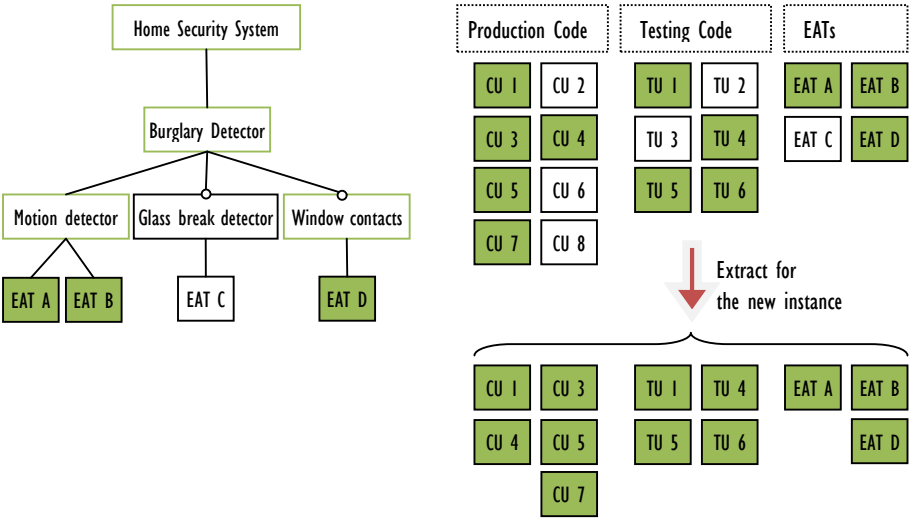


Fig. 8. Using EATs coverage reports to extract artifacts

order to be extended. We used Feature Model DSL as the basis (available online [1]). The tool provides a feature modeling toolbox integrated in the Visual Studio environment. It includes a visual designer to create and modify models. It also provides a configuration window that allows the creation of configurations based on the feature model. We extended the tool in two ways, namely: allow the linkage between features and EATs, and define a course of action to complete the derivation process of individual instances after the configuration process. The remaining of this section will explain the currently available features.

The user can represent features and the relationships between them following the typical feature modeling notation. In our extension of the tool, the leaves of the feature tree can be linked to EATs as shown in Fig. 9.

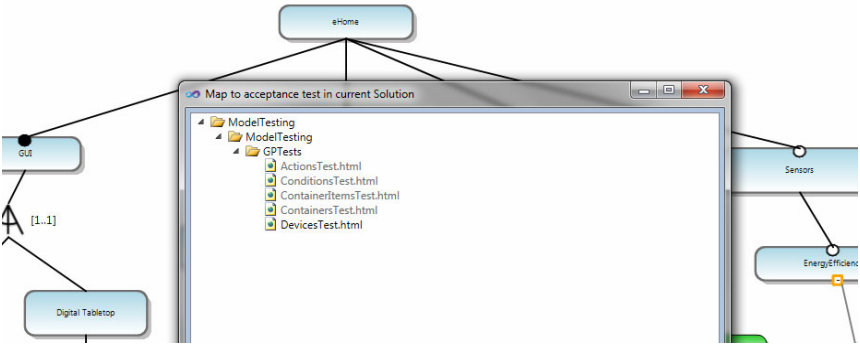


Fig. 9. The leaves of the feature tree can be linked to EATs

The tool also allows the user to run EATs directly from the feature model as shown in Fig. 10. Nodes that have passing tests are coloured in green and those with failing tests are coloured in red. After feature selection, the tool checks the constraints to ensure the validity of the selected subset of features, and it runs only those EATs that are relevant to a given instance. This is shown in Fig. 11. The extended version of the tool will be made available online in Spring 2010.

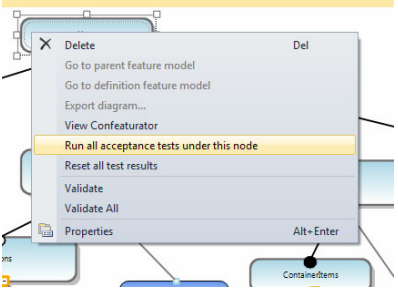


Fig. 10. The tool allows the user to run ATs directly from the variability model

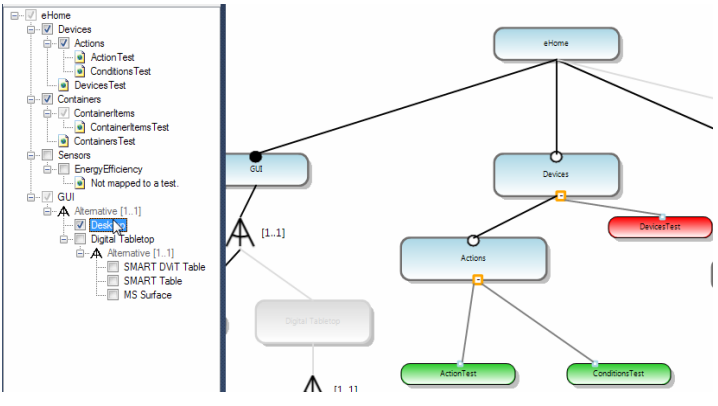


Fig. 11. The tool runs only those EATs that are relevant to a given instance

5 Evaluation

In this paper, we proposed the use of EATs to link feature models to code artifacts. This section presents an evaluation of the proposed approach. We evaluate the approach in two different ways. First, we compare our approach with traditional requirement traceability approaches and other approaches that involve feature models, as discussed in the literature review section. Then, we use the running example presented throughout this paper to list the limitations of the approach (the advantages of the approach were already discussed in the previous section). Using a running

example for validation and evaluation purposes has been a well accepted technique in the community [31, 24, 5].

Table 1 lists a number of criteria against which we conduct our comparison. The criteria are based on guidelines obtained from the literature such as [29] and [2]. The system evolution criterion describes how traceability links are affected with the evolution of a system such as adding or removing requirements. In the case of feature model approaches, we are more concerned with the evolution of variability such as adding or removing variation points and variants. We use the program comprehension criterion to describe the ability of the developers to form a mental model of the variability definition as described in the feature tree as well as the realization of that variability at the code level. This evaluation is limited by the subjectivity arising from the criteria being considered. We intend to conduct a more thorough evaluation to collect empirical evidence of the feasibility and usefulness of the proposed approach.

Having illustrated the advantages of our approach in comparison to other traditional approaches, we think there is a raft of issues that need to be addressed. For one, we cannot currently predict how scalable our approach is – especially when dealing with a large number of variation points and variants. This problem is inherited from the scalability issues associated with feature modeling in general. Furthermore, despite the fact that EATs provide an elegant way to specify functional requirements in software systems, they have not yet been widely used in specifying non-functional attributes such as usability and security (other non-functional attributes like performance can be specified and executed as described in [21]). For feature models that contain variability due to non-functional aspects, our approach may not be sufficient. Moreover, the most common practices involving EATs focus on code artifacts much more than other development artifacts. For organizations that consider design artifacts, for instance, to be essential, the adoption of our approach may result in these artifacts becoming rapidly outdated - mainly because from a developer's perspective there will be no need to maintain them anymore. However, the organization can solve this problem by requiring that some EATs be used as placeholders to associate important information such as links to design documents, standards or data files [23]. Another critical point that may be a real challenge in some organizations is the commitment and discipline needed to provide sufficient EAT coverage of all features in the system in a sustainable manner. Adopting test-driven development practices is one way to deal with this issue.

It is also important to point out that contrary to the initial impression that this approach may lead to architectural drift, the approach may actually improve adherence to the architecture. This is because of the transparency and traceability between the model artifacts and the code artifacts, which provide the developers with a holistic and consistent understanding of the product line. This, however, is still an open issue to investigate in the near future.

Table 1. Comparison between the different approaches of traceability

	Traditional Requirement Traceability	*Traceability through Feature Models*	*Traceability through Feature Models and EAT*
Number of links	Very large, because every requirement is linked to relevant design and code artifacts.	Somewhat large, because every feature is linked to relevant design and code artifacts.	Fairly small, because every feature is only linked to the EATs files specifying that feature.
Quality of links over time	Links become broken or/and outdated without appropriate manual revisions and updates.	Links become broken or/and outdated without appropriate manual revisions and updates.	Links stay consistent and up-to-date because of the immediate feedback on broken or outdated links.
System evolution	Not supported efficiently. If a requirement is added or removed, links have to be re-established.	Not supported efficiently. If a new variation is added or removed, links have to be re-established. Also, there are no automatic checks for new hidden conflicts in the feature model.	Full support. Only links for the added or removed features or variations need to be handled. Failing EATs indicate newly introduced conflicts.
Impact analysis	Provides information on the artifacts that can be potentially impacted by a change. No details on the actual impact.	Provides information on the artifacts that can be potentially impacted by a change. No details on the actual impact.	Provides information on the artifacts that are *actually* impacted by a change, and provides immediate feedback on the *actual* impact of that change.
Program comprehension	Improved over systems with no traceability. But requires an effort for developers to link requirements with code tasks (reading RTMs is not simple). Also, given that variability is not modelled explicitly, handling each type of variation in code is not straightforward.	Reasonable, because requirements are conceptualized at a more comprehendible level of abstraction (i.e. features), and variability is modelled explicitly.	Good, because features are linked directly to code artifacts, and hence variants can be traced to code easily. Also, developers get instant feedback on changes to the code.

6 Conclusion and Future Work

The significance of establishing good traceability links cannot be overstated as evident in the literature and in practical contexts. We presented an approach to link feature models to code artifacts using executable acceptance tests. This paper contributed an approach to provide traceability links in a way that:

- ensures consistency between the feature model and the code artifacts,
- enables the evolution of variability in the feature model, and
- supports the product derivation process.

The valuable implications of these three characteristics were illustrated in detail, and the approach was compared to traditional approaches to highlight its strengths. In spite of the limitations our approach has, we think this is a first yet significant step towards a framework to adopt efficient traceability practices in software product line organizations. For future work, we need to conduct a more comprehensive evaluation of this approach in an industrial setting. We also would like to continue working on a complete tool support for creating, managing, refactoring, and linking EATs within the context of feature models.

References

1. André, F.: Feature Model DSL Homepage (2009), http://featuremodeldsl.codeplex.com/ (accessed February 10, 2010)
2. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. IEEE Transactions on Software Engineering 28(10), 970–983 (2002)
3. Berg, K., Bishop, J., Muthig, D.: Tracing Software Product Line Variability — From Problem to Solution Space. Presented at 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries, White River, South Africa (2005)
4. CaliberRM, http://www.borland.com/us/products/caliber/index.html (accessed March 1, 2010)
5. Cho, H., Lee, K., Kang, K.C.: Feature Relation and Dependency Management: An Aspect-Oriented Approach. In: Proceedings of the 2008 12th international Software Product Line Conference, pp. 3–11. IEEE Computer Society, Washington (2008)
6. Cleland-Huang, J., Zemont, G., Lukasik, W.: A Heterogeneous Solution for Improving the Return on Investment of Requirements Traceability. In: Proceedings of the Requirements Engineering Conference, 12th IEEE international, September 06-10. RE, pp. 230–239. IEEE Computer Society, Washington (2004)
7. DOORS, http://www-01.ibm.com/software/awdtools/doors/ (accessed March 1, 2010)
8. Fey, D., Fajta, R., Boros, A.: Feature Modeling: A Meta-Model to Enhance Usability and Usefulness. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 198–216. Springer, Heidelberg (2002)
9. Filho, I.M., Oliveira, T.C., Lucena, C.J.P.: A proposal for the incorporation of the features model into the UML language. In: Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS 2002), Ciudad Real, Spain (2002)
10. FIT, http://fit.c2.com (accessed November March 1, 2010)
11. Fuchs, N.E.: Specifications are (Preferably) Executable. IEE/BCS Software Engineering Journal 7(5), 323–334 (1992)
12. Ghanam, Y., Maurer, F.: Extreme Product Line Engineering – Refactoring for Variability: A Test-Driven Approach. In: The 11th International Conference on Agile Processes and eXtreme Programming (XP 2010), Trondheim, Norway (2010)

13. Gotel, O., Finkelstein, A.: An Analysis of the Requirements Traceability Problem. In: 1st International Conference on Requirements Eng., pp. 94–101 (1994)
14. GreenPepper, http://www.greenpeppersoftware.com (accessed March 1, 2010)
15. Huang, J.C.: Just enough requirement traceability. In: Proceedings of the 30th Annual International Computer Software and Applications, Chicago, pp. 41–42 (September 2006)
16. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A feature-oriented reuse method with domain specific reference architectures. Annals of Software Engineering 5, 143–168 (1998)
17. Kaner, C.: Cem Kaner on Scenario Testing: The Power of 'What-If...' and Nine Ways to Fuel Your Imagination. Better Software 5(5), 16–22 (2003)
18. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh (1990)
19. Kerievsky, J.: Storytesting, http://industrialxp.org/storytesting.html (accessed March 1, 2010)
20. Manifesto for Agile Software Development, http://www.agilemanifesto.org/ (accessed May 13, 2010)
21. Marchetto, A.: http://selab.fbk.eu/swat/slide/2_Fitnesse.ppt (accessed March 10, 2010)
22. Melnik, G., Maurer, F., Chiasson, M.: Executable Acceptance Tests for Communicating Business Requirements: Customer Perspective. In: Proc. Agile 2006 Conf., pp. 35–46. IEEE CS Press, Los Alamitos (2006)
23. Park, S.S., Maurer, F.: The benefits and challenges of executable acceptance testing. In: APOS 2008: Proceedings of the 2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral, pp. 19–22 (2008)
24. Parra, C., Blanc, X., Duchien, L.: Context Awareness for Dynamic Service-Oriented Product Lines. In: Proceedings of 13th International Software Product Line Conference (SPLC), San Francisco, CA, USA (2009)
25. Pashov, I.: Feature Based Method for Supporting Architecture Refactoring and Maintenance of Long-Life Software Systems. PhD Thesis, Technical University Ilmenau (2004)
26. Perry, W.: Effective Methods for Software Testing, 2/e. John Wiley & Sons, New York (2000)
27. Pure::Systems, http://www.pure-systems.com/DOORS.102+M54a708de802.0.html (accessed March 1, 2010)
28. Ramesh, B., Jarke, M.: Toward Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering 27(1), 58–93 (2001)
29. Riebisch, M.: Supporting Evolutionary Development by Feature Models and Traceability Links. In: Proceedings of the 11th IEEE International Conference and Workshop on Engineering of Computer-Based Systems, ECBS, May 24-27, p. 370. IEEE Computer Society, Washington (2004)
30. Riebisch, M.: Towards a more precise definition of feature models. In: Riebisch, M., Coplien, J.O. (eds.) Modelling Variability for Object-Oriented Product Lines (2003) (Position Paper)
31. Tun, T.T., Boucher, Q., Classen, A., Hubaux, A., Heymans, P.: Relating Requirements and Feature Configurations: A Systematic Approach. In: International Software Product Line Conference (SPLC 2009) (2009)

Avoiding Redundant Testing in Application Engineering

Vanessa Stricker, Andreas Metzger, and Klaus Pohl

Paluno (The Ruhr Institute for Software Technology)
University of Duisburg-Essen, 45127 Essen, Germany
{Vanessa.Stricker,Andreas.Metzger,Klaus.Pohl}@sse.uni-due.de

Abstract. Many software product line testing techniques have been presented in the literature. The majority of those techniques address how to define reusable test assets (such as test models or test scenarios) in domain engineering and how to exploit those assets during application engineering. In addition to test case reuse however, the execution of test cases constitutes one important activity during application testing. Without a systematic support for the test execution in application engineering, while considering the specifics of product lines, product line artifacts might be tested redundantly. Redundant testing in application engineering, however, can lead to an increased testing effort without increasing the chance of uncovering failures. In this paper, we propose the model-based ScenTED-DF technique to avoid redundant testing in application engineering. Our technique builds on data flow-based testing techniques for single systems and adapts and extends those techniques to consider product line variability. The paper sketches the prototypical implementation of our technique to show its general feasibility and automation potential, and it describes the results of experiments using an academic product line to demonstrate that ScenTED-DF is capable of avoiding redundant tests in application engineering.

Keywords: Software product line testing, application engineering, data flow, regression testing.

1 Introduction

Software product line engineering (SPLE) has proven to be a successful paradigm for developing a diversity of similar software products at low costs, in short time, and with high quality [1]. SPLE is based on the planned, systematic, and pro-active reuse of development artifacts (including requirements, components, and test cases). To this end, two processes are differentiated: In *domain engineering*, the commonality and the variability of the products of the product line are defined and the reusable artifacts are realized. In *application engineering* customer-specific products are derived from the reusable artifacts. Application engineering exploits the variability of the reusable artifacts by binding (resolving) variability according to the requirements defined for the particular product.

1.1 Problem Statement

Several techniques for SPL testing exist that advocate the early test of product line artifacts during domain engineering (see [2, 3, 4]). Although such an early test can

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 226–240, 2010.

uncover critical problems, two problems remain. Firstly, due to the variability of the reusable artifacts, it is impossible (except for trivial product lines) to comprehensively test all products in domain engineering (see [4]). Secondly, in cases where specific variants are only developed based on concrete customer demand, not all artifacts needed for testing a product might be available during domain engineering. Thus, testing in application engineering remains key for SPL testing. Many techniques have been proposed to efficiently support application testing by deriving test cases from reusable artifacts developed in domain engineering (see Section 2).

However, in addition, the *execution* of test cases constitutes one important activity during application testing. Without techniques that systematically support the test execution in application engineering while considering the specifics of SPLE, product line artifacts might be tested redundantly – without increasing the overall test coverage. Redundant testing can occur if two products, in addition to the commonality, share similarities; e.g., if the same variant V1 is bound in product line products p_1 and p_2. If V1 is *independent* of other variants and has been tested in p_1 given a pre-defined coverage criterion, testing V1 again for p_2 using the same coverage criterion will typically not increase the chance of uncovering defects in V1.

Summarizing, without systematic support for avoiding redundant testing in application engineering, testing effort might be invested without additional benefits.

Addressing this problem for simple, structural test coverage criteria, such as branch coverage, is straightforward. One only needs to trace the branches that have been tested for p_1 and has to determine the additional branches that have thus to be covered in p_2. However, branch coverage has been observed to be a too weak criterion during testing. For instance, Rapps and Weyuker [5] have already stated in 1982 that – for single software systems – a purely structural test of the software based on control flow, is not sufficient and thus proposed considering the data flow in addition.

For software product lines, the weakness of the structural criteria is even bigger as those structural criteria are not capable of grasping the dependencies between variants of an SPL. As an example, let us assume that a variant V2 modifies the data that is used by a variant V1 (e.g., additional data manipulations are introduced by V2). Let us further assume that p_1 only binds V1 but that p_2 binds V1 and V2. In this case, only testing V2 in p_2 (and relying on the already tested V1 in p_1) is not enough, as the abovementioned data dependencies would not be tested.

Summarizing, if branch coverage was used as a criterion to avoid redundant testing in application engineering, critical dependencies could go untested, leading to a high probability that failures remain uncovered.

1.2 Solution Idea and Contribution of the Paper

In this paper, we propose the model-based ScenTED-DF technique to avoid redundant testing in application engineering. ScenTED-DF builds on *data flow-based testing techniques* for single systems and adapts and extends those techniques to consider product line variability. Based on a data flow-based coverage criterion, the test coverage of previously tested SPL products are employed in order to identify the paths (data flow) that have to be re-tested for the new SPL product.

The remainder of the paper is structured as follows: Section 2 describes how ScenTED-DF progresses from the state of the art. Section 3 introduces the fundamentals

for our technique. Section 4 describes the key concepts and steps of ScenTED-DF, illustrated by a small example. In Section 5 we present the evaluation of our technique, which includes the prototypical implementation to show the general feasibility and automation potential of the technique, as well as experiments using an academic product line to demonstrate that ScenTED-DF is capable of avoiding redundant tests in application engineering. Section 6 critically discussed the results and provides an outlook to future work.

2 Progress from State of the Art

Many product line testing techniques have been presented in the literature (see [3, 2]). Following the pro-active reuse concept underlying SPLE [1], the majority of those techniques addresses how to define reusable test assets (such as test models or test scenarios) in domain engineering and how to exploit those assets during application engineering. Representatives for those techniques are the one from Bertolino and Gnesi [6], Geppert et al. [8], McGregor et al. [7], and Nebut et al. [9]. In our previous work, we have developed the ScenTED technique. ScenTED facilitates the systematic, requirements-based derivation and reuse of test cases for system, integration and performance testing in SPLE based on control flow [4] (figure 1 in section 3 illustrates the difference between control and data flow based testing using an abstract example). All those techniques avoid redundant testing activities by exploiting reuse during test case generation. However, those techniques do not address the problem (as introduced above) that the execution of those test cases during application engineering can still mean that redundant testing activities are performed.

In [1], the problem of redundant testing activities during application engineering is observed. Based on a cost model, the gains of reusing test results for the common parts of the product line are estimated. However, reusing the test coverage from previous products is not considered as a further concept to avoid redundant activities.

In [10] an approach is introduced that aims at discovering the commonality among execution traces customer-specific products of a product line to reduce test execution of each new product. During each test of a new product, the current trace is checked against the set of common traces and if the current trace has already been covered in a previous test, this trace is used to replace the actual execution. In contrast to that technique, ScenTED-DF is able to also reuse partial traces (i.e., paths) from previous tests, as data dependencies are explicitly considered. This means that our technique provides a higher reuse potential than the one based on complete execution traces. In addition, the reuse gain of our technique has been experimentally evaluated.

One testing technique from single systems engineering that bears a relation to the problem addressed in this paper is *regression testing* [11]. Regression testing is performed to assess that no defects are introduced in a new version of a software product. To reduce the effort of regression testing, usually an impact analysis is performed to identify the test cases that have to be re-executed for the new version of the product. As a significant difference, the different products of a software product line cannot be considered as different versions of the same product. Most notably, the SPL products significantly differ in terms of bound features (variants). This firstly means that not all test cases from one SPL product are relevant for another SPL product and secondly

that additional test cases might be necessary to cope with the differences between the products in term of the bound variants.

Summarizing, there currently exist no techniques that systematically address the problem of avoiding redundant test activities between the products of a software product line, while considering data flow dependencies. ScenTED-DF, which is introduced in this paper, addresses this gap.

3 Fundamentals: Data Flow-Based Testing of Single Systems

Data flow-based testing techniques focus on finding failures in the system due to *data dependencies* along execution paths. A data dependency exists if an action (e.g., assignment, statement, condition) in a software system reads the value of a data object (e.g., local or global variable) that has been manipulated by a previous action on the execution path. Figure 1a shows the control flow graph for a very simple program and highlights the data dependencies that occur in this program. To understand the significant difference from structural testing, Figure 1a also shows the two paths that are enough to achieve branch coverage. As can be seen, data flow-based testing would consider an additional path to cover the remaining data dependency.

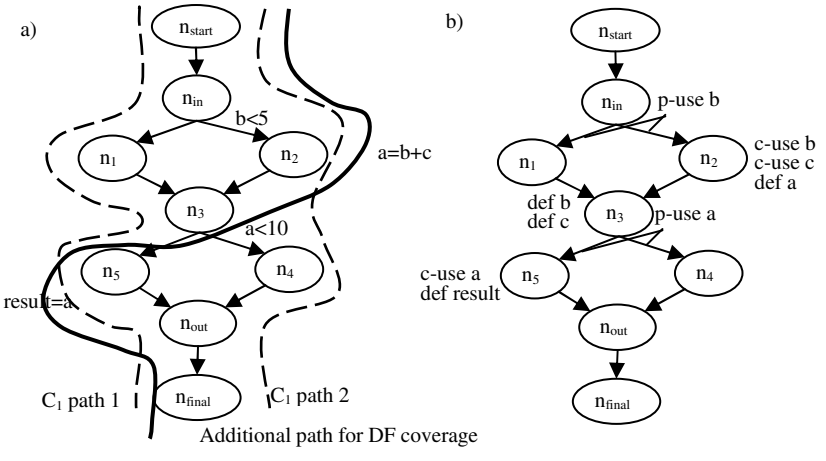


Fig. 1. Control and data flow coverage in an abstract control flow graph

One widely used data flow-based test technique is the defs-uses-test [5]. It has been developed for *control flow graphs* which are derived from the program code of a software system. For each node of the control flow graph all accesses to variables are annotated. *Writing* accesses are the so called *defs* and *reading* accesses the so called *uses*. The *uses* are distinguished into *c-uses* for *computational* accesses in expressions and into *p-uses* for *predicate* accesses in conditional statements. Figure 1b shows how a control flow graph is annotated with defs and uses.

The main idea behind the defs-uses-test is to systematically determine the dependencies between the writing accesses (defs) of each variable *x* and the reading accesses (uses) for that variable *x*. All paths from each action annotated with a *def x* to each

action annotated with a *use x* are identified as a data dependency, provided that the paths are *definition free*, i.e., that there is not further writing accesses in any of the actions between the def x and use x. All such data dependencies need to be covered by the set of test cases to achieve 100% coverage of the system under test based on the data flow. This is known as the *all-du-path coverage criterion*.

4 ScenTED-DF: Data Flow-Based Test for Product Lines

This section introduces ScenTED-DF, our data flow-based test technique that allows avoiding redundant testing in application engineering. ScenTED-DF uses the defs-uses-test as a foundation (see Section 3). This section explains how the defs-uses-test is adapted towards product line context. First, we describe the test models that we use for ScenTED-DF (Section 4.1). Second, the steps of the ScenTED-DF technique are introduced (Section 4.2).

4.1 Data Flow-Based Test Model with Variability

Similar to our ScenTED technique [4], ScenTED-DF is a model-based testing technique. ScenTED uses activity diagrams as test models to specify the control flow of customer-specific products. Accordingly, we augment those activity diagrams with def and use statements (see Section 3), in order to incorporate the concepts needed for the defs-uses-test.

ScenTED has introduced variable activity diagrams, i.e., activity diagrams that consider the variability of the product line. In variable activity diagrams *variation points* and *variants* are explicitly specified by relating elements of the activity diagram to an orthogonal variability model (OVM) [1]. The OVM contains all variation points of a SPL as well as their variants together with any constraints[1]. A small OVM and how it is related to an activity diagram is shown in Figure 2 for an excerpt of an eShop product line (the registration of an eShop buyer).

Augmenting those variable activity diagrams with defs and uses is straightforward, as the concepts used in activity diagrams can be mapped to concepts of control flow graphs (used in the initial defs-uses-test): The actions of an activity diagram are mapped to nodes of the control flow graph. The decision- and merge- nodes of an activity diagram are mapped to the decision nodes of a control flow graph. Accordingly, defs and uses are annotated to the actions and decision nodes in the activity diagram as it is done for the nodes of a control flow graph. Figure 2 shows how defs and uses are annotated in the activity diagram.

4.2 Steps of ScenTED-DF

As mentioned above, the steps of ScenTED-DF are separated into those performed in domain engineering and those performed in application engineering. The steps in domain engineering are the preparation for the actual defs-uses-test in application

[1] The approach is not dependent on the usage of OVM. Other approaches that allow the definition of relations between the variability and parts of the activity diagram can also be used, e.g. feature diagrams [12].

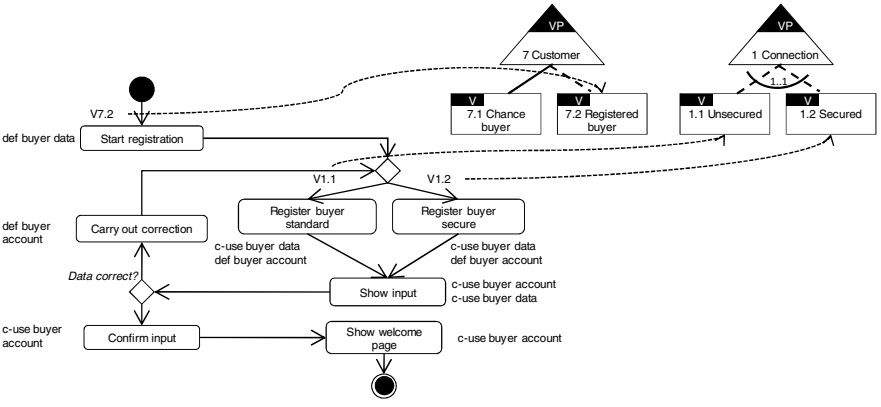


Fig. 2. Variable activity diagram *Register buyer* annotated with data flow

engineering for different products $p_1 \ldots p_{k-1}$. This allows to exploit the capabilities of SPLs and to reuse certain artifacts created in domain engineering that contain the commonalities and consider the variability explicitly in application engineering. The steps are described in detail below.

4.2.1 Steps in Domain Engineering

D1: Creation of data flow test model

As described in Section 4.1 ScenTED-DF uses activity diagrams as test model. Thus, the first step in preparation for the data flow-based test which is performed in domain engineering is the annotation of those activity diagrams with data accesses. The data flow attributes are added for all data objects that are read or written by the respective action. This step is a creative modeling activity which cannot be automated. However, it can be supported by appropriate modeling tools.

D2: Identification of variable data dependencies

Ideally, as mentioned in Section 3, all data dependencies would be covered by the defs-uses-test (all-du-path coverage). However, p-uses currently cannot be considered in the ScenTED-DF technique, as the usage of data objects at decision nodes that coincide with variation points is an issue to ongoing research. Thus, ScenTED-DF resorts to the *all-c-uses criterion* (which guarantees 100% coverage of all data dependencies between defs and c-uses) and introduces additional steps (see D3 and A3) to ensure complete branch coverage (as a minimal coverage criterion).

Using the all-c-uses criterion, all data dependencies between writing accesses and all possible reading accesses are computed in domain engineering, i.e., the set of all data dependencies that might occur in all customer-specific products is computed. The data dependencies that are relevant for a specific product are selected from this set in application engineering (steps A1-A3). In ScenTED-DF data dependencies are noted as a tuple of three elements: (data object, def, uses). Two different types of dependencies between the data dependencies and the variants need to be considered:

1) A variant can be required for a data dependency to exist, i.e., the writing or reading access that constitutes the data dependency is performed in an action that is associated to a variant. e.g. (Buyer data, Start registration, Register buyer standard) requires the binding of V1.1 and V7.2 in Figure 2.
2) Data dependencies can be associated indirectly with variants; e.g. in cases where the data dependency occurs on several paths which have branches and actions that are associated to different variants. The data dependency (Buyer data, Start registration, Show input) in Figure 2 for example depends on the selection of variant V7.2 but is independent from the selection of V1.1 or V1.2.

Data dependency	**Relation to variants**		
(data object, def, c-use)	V1.1	V1.2	V7.2
(BuyerData, startRegistration, registerBuyerStandard)	1	*	1
(BuyerData, startRegistration, registerBuyerSecure)	*	1	1
(BuyerData, startRegistration, showInput)	*	*	1
(BuyerAccount, registerBuyerStandard , showInput)	1	*	1
(BuyerAccount, registerBuyerStandard, confirmInput)	1	*	1
(BuyerAccount, registerBuyerStandard, showWelcomePage)	1	*	1
(BuyerAccount, registerBuyerSecure, showInput)	*	1	1
(BuyerAccount, registerBuyerSecure, confirmInput)	*	1	1
(BuyerAccount, registerBuyerSecure, showWelcomePage)	*	1	1

Fig. 3. Data dependency matrix for Register buyer

A d*ata dependency matrix* is exploited to store the identified dependencies to the different variants of the SPL. Based on the information annotated to the activity diagram (see Section 3.1), this step can be automated. Figure 3 shows the data dependency matrix for the activity diagram shown in Figure 2, in which a required variant (as described above) is indicated by '*1'* and variants that do not impact on data dependencies by '**' (don't care).* In cases where a variant must not be selected, this is indicated by '0' (not visible in the example). The computation of the matrix avoids that the identification of data dependencies has to be performed for each derived product and also allows for the automated identification of all relevant dependencies for a newly derived product in application engineering (see A1).

D3: Ensuring branch coverage
As we currently do not support p-uses (see above), we added an additional criterion to guarantee adequate test coverage. Therefore, a *traceability matrix* is introduced which associates all branches to the different variants, such that the branch coverage can be traced for the different variants. This association can be computed using the OVM.

4.2.2 Steps in Application Engineering

A1: Identification of data dependencies to be covered for a customer-specific product
The dependency and traceability matrices defined in domain engineering are reused for the test of a new derived product p_k. When a new product is derived, the data dependencies of interest have to be determined. For this step, the dependency matrix is used as well as the information about the variants that have been bound in the product. The dependencies can be identified by checking the columns of the dependency

matrix, which contain the bound variants of the product. Wherever a dependency is marked (either as required or don't care) in any of those columns, the row has to be considered, otherwise the row can be ignored. In the same way all columns that are marked required for a variant that is not selected have to be ignored, since a data dependency can depend on more than just one variant. The remaining rows contain the relevant data dependencies, which are analyzed in the next steps. For a product p_k that binds the variants V1.1 and V7.2 as shown in Figure 3, the relevant dependencies are:

dep_k={(BuyerData, startRegistration, registerBuyerStandard), (BuyerData, startRegistration, showInput), (BuyerAccount, registerBuyerStandard, showInput), (BuyerAccount, registerBuyerStandard, confirmInput), (BuyerAccount, registerBuyerStandard, showWelcomePage)}

The set dep_k includes the very dependencies that have to be considered to achieve full all-c-use coverage for product p_k. However, the main motivation of ScenTED-DF is to reduce test effort in application engineering by avoiding redundant tests with respect to the data flow coverage. Accordingly, the main outcome of step A1 is the computation of the set of data dependencies that actually have to be tested with new test cases in order to achieve full test coverage for p_k. This means that it has to be identified which of the data dependencies relevant for the product p_k are not covered by the set of data dependencies $dep_{1..k-1} = U_{i=1..k-1}$ dep*i* hat have successfully been tested for other products $p_1 \ldots p_{k-1}$. Those dependencies can be identified by computing the difference (delta) of the two sets: $delta_k = dep_k \setminus dep_{1..k-1}$. According to the idea underlying the ScenTED-DF technique, testing dependencies that already have been tested in other products would increase the testing effort without increasing the probability of uncovering failures. Thus, the intersection of $dep_k \cap dep_{1..k-1}$ constitutes the amount of data dependencies that are already tested and for which redundant test case execution can be avoided.

A2: Defining and executing new test cases for all untested data dependencies
To guarantee full test coverage, a set of test cases has to be determined that covers all elements of $delta_k$. The derivation of the set of test cases DT_k adheres to the all-c-use criterion which is defined in the defs-uses-test (see Section 2.3.2). The step of test cases derivation cannot be automated. Neither ScenTED nor ScenTED-DF or the original defs-uses-test addresses the derivation of concrete test data that are needed to specify test cases. This is in general a research topic on its own, since the derivation of test data has huge influence on the quality of the test and often involves high effort as in many cases it is done "manually". However, we expect, that avoiding redundant test coverage also means that we can avoid deriving test cases that would only lead to redundant testing.

A3: Ensuring branch coverage
Since the all-c-uses criterion does not assure branch coverage (as described for D2), a set of additional test cases BT_k is derived and executed to complement DT_k. This step aims at ensuring that the set of all branches $b \in B_k$ of p_k which are not covered by the test cases in DT_k is sufficiently covered. In order to avoid redundant branch coverage, ScenTED-DF uses the coverage of the branches that have already been tested in $p_1 \ldots p_{k-1}$ to achieve a full branch coverage based for p_k.

This means that if the product under test is not the first product p_1 of a SPL, but the products $p_1 \ldots p_{k-1}$ have already been tested, it has to be identified which branches $b \in$

of B_k are already covered. This can be done by using the traceability matrices and the binding information. If the variants that have been associated with the branches $b \in B_k$ have already been bound in $p_1 \ldots p_{k-1}$, complete branch coverage for this variant is assured. For the remaining branches new test cases have to be derived.

After all test cases, for the data dependencies and branches, have been executed successfully, it is important to store the information such as the binding information, the tested dependencies, the test cases used to perform the complete test in such a way, that these information can be taken into account for the test coverage of further products.

5 Evaluation and Discussion

The main motivation for ScenTED-DF is to avoid, as far as possible, redundant testing in application engineering, while guaranteeing the expected data flow-based test coverage for the products of a SPL. This is achieved by exploiting the knowledge about the variability and commonalities of a SPL and thus the similarities between the different products of a SPL. This section presents an experimental evaluation of our technique using an academic SPL. This SPL has been used in the past for several experiments related to SPL testing. Before we introduce the SPL and the experimental results, we summarize the tool support used during the experiments.

5.1 Prototypical Implementation

A tool prototype has been implemented to support the execution of the ScenTED-DF technique presented above. The tool consists of two main components: (1) a graphical editor which allows creating activity models annotated with data usage, and (2) implementations of the algorithms that perform the different steps of ScenTED-DF in domain and application engineering.

The graphical editor has been realized using the *Eclipse Modelling Framework (EMF)* and the *Graphical Modelling Framework (GMF).* The editor allows modelling activity diagrams augmented with variation points and variants (as introduced for the original ScenTED technique), as well as defs and c-uses annotations (needed for ScenTED-DF). Thereby, it supports step D1 (see Section 4.2.1).

The activity diagrams created using the graphical editor are used in the second component of the tool prototype. This tool component performs an automatic computation of the data dependency and traceability matrices in domain engineering (steps D2 and D3, see Section 4.2.1). In application engineering (step A1), the relevant data dependencies are determined and the delta between new data dependencies and previously tested ones is automatically computed. Step A2 is not automated nor supported as described in Section 4.2.1. Step A3, which aims at assuring branch coverage, can has not been implemented in the tool prototype.

5.2 Example Product Line

The evaluation of our technique is based on a configurable *eShop Platform* which allows deriving an eShop product according to the customers' (i.e., shop owners') needs so that the goods can be sold to buyers via the Internet. This is an academic

product line which has been developed to evaluate the various testing activities of the ScenTED technique [13]. To use it for an evaluation of ScenTED-DF, it has been extended with data flow annotations (i.e., defs and uses). Due to the size of the SPL, we can only include the variability model (OVM) of the SPL in this paper, which is shown in Figure 4.

The customer who likes to have an eShop to run his business can choose between 17 variants for 9 variations points addressing the different types of Buyers (VP7), Payment options (VP 3), Data storage alternatives (VP 9), etc. The minimum number of variants that has to be selected in a customer-specific product is 6 while the maximum number of variants that can be chosen in one product is 14. As it can be seen, some of the variation points like the payment by Card are optional and do not have to be included in any product. If they are included however, they impose certain constraints on the selection of other variants. Considering all 17 variants that can be bound from the 9 variation points as well as their constraints and dependencies over 1500 different valid products can be derived from the eShop SPL (cf. [14]).

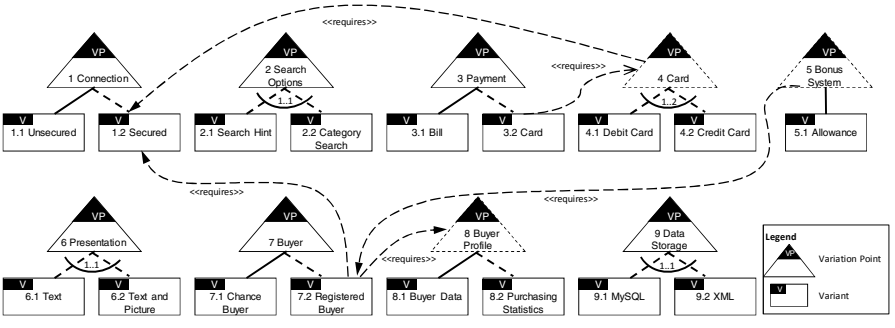


Fig. 4. Variability model of the eShop SPL

5.3 Experimental Evaluation

The eShop has been used to perform an experimental evaluation of ScenTED-DF. The goal of this evaluation was to analyse in how far ScenTED-DF is able to avoid redundant testing by identifying already covered data dependencies. In particular the following GQM-goal of the evaluation has been addressed:

Goal: Analyse the ScenTED-DF technique for the purpose of better understanding the potential of avoiding redundant tests in application engineering from the viewpoint of the researcher in the context of the eShop SPL.

In the following, the design of the experimental evaluation will be explained, as well as the results and their critical discussion.

5.3.1 Design of the Experimental Evaluation

First of all, the activity diagrams describing the eShop SPL have been modelled using the prototypical tool. Furthermore, they have been annotated with data usages (Step D1). The prototype has been used to automatically compute the data and traceability matrices (Step D2 and D3).

Since ScenTED-DF does not make any assumptions about the derivation order of the products but assumes that they are tested in the order as requested by customers, this unpredictability was to be considered in the evaluation of the technique. Therefore, different sets of products (in different order) have been derived randomly. More precisely, 20 sets, each of which containing the binding information for 20 products p_k, have been derived in a random order.

For this random derivation, the number of variants that are bound in each of the derived products has been determined randomly. As has been explained above, a minimum number of 6 and a maximum number of 14 variants can be selected during this step. After picking a random number in the range of 6 to 14, the concrete variants to be bound have been selected randomly from the OVM model[2].

For each of those sets and each of the products, the ScenTED-DF technique has been applied. The data flow coverage and the delta to already tested products have been measured and are presented in section 5.3.2.

It should be noted that, so far, we did not derive concrete test cases and thus did not execute test cases during the experiments. Instead, we assumed that appropriate tests are executed in order to cover all data dependencies that have been identified to be relevant for a product p_k. Thus, we can only measure the relative data-flow coverage. In order to provide a basis to compute the actual efficiency gained by the application of the technique, concrete test cases need to be derived using a test technique used for single system development for each derived product of the eShop. We are planning to perform this exercise in future work.

5.3.2 Results

During the evaluation of the ScenTED-DF technique we have been interested mainly in two measures regarding the data flow coverage: (1) The number of the relevant data dependencies for a product p_k that have already been covered by earlier tested products $dep_k \cap dep_{1..k-1}$ (see Section 4.2.2); (2) the overall number of data dependencies in the eShop product line that have been covered by customer-specific products during testing ($dep_{spl} \cap dep_k$) has been analysed.

The computation performed in step D2 revealed that not all data dependencies of the eShop product line are affected by variability. Therefore, it is expected that a significant amount of effort can be saved during the test execution as these non-variable dependencies only have to be tested in the first product p_1. Afterwards, the affected data dependencies can be assumed to be covered in further products $p_2 \ldots p_k$. The tool prototype identified 58 different dependencies in the eShop SPL. However, 13 of these dependencies are found twice because they can be found on different paths in the activity diagrams (see Section 4.1). Furthermore, 38 dependencies are non-variable, i.e. those dependencies are not affected by any variant. That means that the test results of approx. 65% of all possible dependencies will already be covered after p_1 independent of the binding information.

The examination of the 20 sets of customer-specific products has shown that the amount of covered data dependencies strongly varies depending on the order of product derivation and strongly depends on which variants are bound. Figure 5a shows the number of products that have been tested in the different sets in order to achieve

[2] Randomization has been realized using java.util.random.

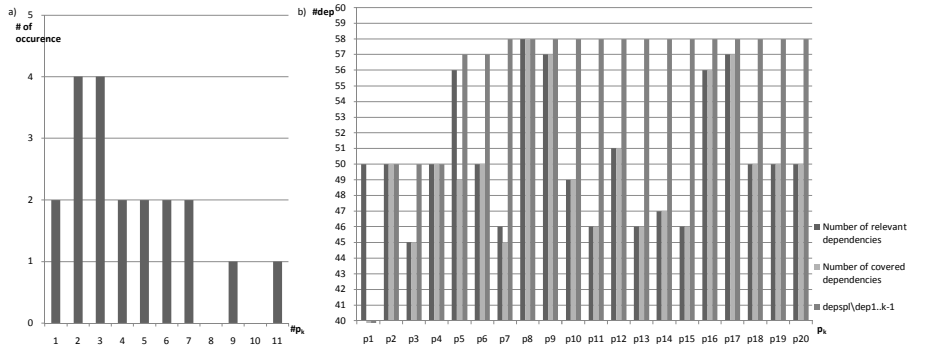


Fig. 5. Results of the experimental evaluation

100% coverage of all 58 data dependencies that occur in the eShop SPL (dep_{spl}). It can be seen that the number of products that need to be tested ($\#p_k$) to reach 100% data flow coverage range from one single product (in two occurrences) to 11 products (in one occurrence).

The coverage of the overall data dependencies in the SPL for one of the randomly generated sets of products is shown in Figure 5b ($dep_{SPL} \setminus dep_{1..k-1}$). It can be seen that after the initial product p_1, 3 further customer-specific products $p_2...p_4$ are derived which all benefit from the data flow coverage of p_1. For those products no new dependencies are relevant and thus p_2 to p_4 achieve 100% data flow coverage. For p_5 new dependencies become relevant that are not covered by p_1 to p_4 and thus new test cases have to be derived and executed to assure complete data flow coverage for this product. After p_7, however, all data dependencies of the eShop have been tested.

Figure 5b also shows for each product p_k a bar displaying the number of relevant data dependencies and a bar displaying how much of that data dependencies are already covered by the test of earlier products. It can be seen that after p1 for example 50 data dependencies have been tested. The bar for p_2 shows that also for this product 50 data dependencies have to be covered. Since these are the same as those that have already been tested 100% data flow coverage is given for p_2. The bar for p_3 shows, that only 45 data dependencies are relevant for p3 so that not even all of the covered data dependencies are needed to achieve 100% data flow coverage for the product.

5.3.3 Discussion of Results

Of course, not all of the 1500 products (in our *eShop* example) will be derived in reality and thus testing them would involve too much effort in respect to the benefits gained from an exhaustive test. However, ScenTED-DF assumes that with the continuous derivation of products the data dependency coverage eventually increases and thus helps to avoid redundant tests. In fact, the evaluation above has provided first evidence in favour of this assumption.

Since we start from the outset that for the SPL under test a data flow-based testing techniques is used, ScenTED-DF does not impose additional effort to prepare the testing, as the required test models including data flow attributes are created anyways and all other steps till the test case generation are performed automatically. Instead

effort can be saved considering the characteristics of SPLs in which the derived products often will share many variants (similarities) and therefore also many of the related data flow dependencies are shared. Thus, only the new data dependencies have to be tested while the already covered data dependencies do not have to be tested again. In some cases, the new variants won't even affect the dependencies, such that the data flow of a new product is already completely covered by past tests.

The results presented in 5.3.2 demonstrate that, ScenTED-DF offers the potential to save testing effort, since a certain amount of data dependencies do not have to be tested again, as they are already covered in the test of earlier products. The amount of "reuse" is promising and thus we expect increased efficiency when running real test cases.

Of course, these results should be interpreted carefully. The amount of already tested data flow coverage that is gained in a SPL has been measured as the sum of all tested products. Considering the internal validity however, this measure cannot directly be used to reason about the effort that is needed or can be saved for the test of a new customer-specific product. The technique does not prescribe the way test cases are derived and a test case can cover more than one dependency and certainly more than one branch. Thus, the number of dependencies that do not have to be tested does not directly correlate with the number of test cases (derivation and execution) that do not have to be executed again. It could be possible that, with good test case design, the test cases covering the new dependencies also cover the already tested dependencies. In this case there would be no saved effort in comparison to a test without avoiding the redundant coverage. Furthermore, addressing the construct validity, as described in section 5.3.2 the actual efficiency of the technique cannot be measured without performing a complete single system test for each derived product, which could also pose a threat to the conclusion validity.

Finally, the practical relevance of the results should be supported by real-life or at least realistic SPLs, as the eShop is a relatively small and academic example. It constitutes an extension of an earlier model with data flow annotations. Although the data flow annotation to the product line has been performed as objective as possible, certain influence from the researcher on the resulting data dependencies cannot be totally excluded. Thus, this poses a threat to external validity. For instance, the order of products can have a much more severe impact on the number of covered data dependencies than it already had in the eShop example. However, considering the empirical reliability, we assume that the experimental evaluation can be repeated in different settings using the ScenTED-DF technique and will most likely reveal similar results.

6 Conclusion and Perspectives

This paper has introduced the model-based ScenTED-DF technique that allows avoiding redundant testing in application engineering. The technique uses the defs-uses-test technique and adapts and extends it to consider product line variability. Based on a data flow-based coverage criterion, the test coverage of previously tested SPL products are employed in order to identify data dependencies that are covered and do not have to be tested again. An experimental evaluation has been performed on an academic SPL example which revealed that the technique is able to exploit covered data

dependencies from already tested products to ensure complete data flow coverage without testing redundantly. Performance has not been a relevant aspect in this experimental evaluation. However, for larger SPLs a performance analysis of the presented algorithms should be performed.

A major shortcoming of the presented technique in its current status is that p-uses are not supported since the usage of data objects in variation points is not formalized, yet. The ScenTED technique does not make any statements about how conditional expressions are to be annotated to variation points at which more than one variant can be bound requiring (which would require a decision node in the customer-specific product). Due to this lack of support for p-uses, stronger data flow-based coverage criteria (such as all-uses) currently cannot be used within ScenTED-DF. Thus, future work will focus on annotating variable activity diagrams with conditional expressions and on extending ScenTED-DF as well as the prototypical implementation for the support of p-uses. With this extended technique a more extensive evaluation using a more complex SPL from an industrial setting is planned.

Acknowledgments. This work has been funded by the DFG under grant PO 607/2-1 IST-SPL.

References

1. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering – Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
2. Lamancha, B.M., Usaola, M.P., Velthius, M.P.: Software Product Line Testing: A Systematic Review. In: Shishkov, B., Cordeiro, J., Ranchordas, A. (eds.) Proceedings of the 4th International Conference on Software and Data Technologies (ICSOFT), vol. 1, pp. 23–30. INSTICC Press (2009)
3. Tevanlinna, A., Taina, J., Kauppinen, R.: Product Family Testing – a Survey. ACM SIGSOFT Software Engineering Notes 29(2) (2004)
4. Pohl, K., Metzger, A.: Software Product Line Testing: Exploring Principles and Potential Solutions. Communications of the ACM 49(12), 78–81 (2006)
5. Rapps, S., Weyuker, E.J.: Data Flow Analysis Techniques for Test Data Selection. In: Proceedings of 6th International Conference on Software Engineering (ICSE 1982), pp. 272–278. IEEE Computer Society, Los Alamitos (1982) Catalog No. 82CH1795-4
6. Bertolino, A., Gnesi, S.: Use Case-Based Testing of Product Lines. In: Paakki, J., Inverardi, P. (eds.) Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), Poster Session, pp. 355–358. ACM Press, New York (2003)
7. McGregor, J.D.: Testing a Software Product Line. Software Engineering Institute, Technical Report, CMU/SEI-2001-TR-022, ESC-TR-2001-022, Product Line Systems Program. Carnegie Mellon University (2001)
8. Geppert, B., Li, J., Roessler, F., Weiss, D.M.: Towards Generating Acceptance Tests for Product Lines. In: Bosch, J., Krueger, C. (eds.) ICOIN 2004 and ICSR 2004. LNCS, vol. 3107, pp. 35–48. Springer, Heidelberg (2004)
9. Nebut, C., Fleurey, F., Le Traon, Y., Jézéquel, J.: A Requirements based Approach to Test Product Families. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 198–210. Springer, Heidelberg (2004)

10. Li, J.J., Geppert, B., Roessler, F., Weiss, D.M.: Reuse Execution Traces to Reduce Testing of Product Lines. In: SPLIT Workshop, Proceedings of the 11th International Product Line Conference (SPLC), Second Volume (Workshops), pp. 65–72. Kindai Kagaku Sha Co. Ltd., Tokyo (2007)
11. Li, Y., Wahl, N.J.: An Overview of Regression Testing. ACM SIGSOFT Software Engineering Notes 24(1), 69–73 (1999)
12. Metzger, A., Heymans, P., Pohl, K., Schobbens, P.-Y., Saval, G.: Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In: Sutcliffe, A., Jalote, P. (eds.) Proceedings of the 15th IEEE Intl. Conference on Requirements Engineering (RE 2007), pp. 243–253. IEEE Computer Society, Los Alamitos (2007)
13. Metzger, A.: Testing in a Software Product Line. Panel Presentation at the 10th International Software Product Line Conference (SPLC) (2006)
14. Maßen, T.v.d., Lichter, H.: Determining the Variation Degree of Feature Models. In: Obbink, J.H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 82–88. Springer, Heidelberg (2005)

Improving the Testing and Testability of Software Product Lines

Isis Cabral, Myra B. Cohen, and Gregg Rothermel

Department of Computer Science,University of Nebraska-Lincoln
{icabral,myra,grother}@cse.unl.edu

Abstract. Software Product Line (SPL) engineering offers several advantages in the development of families of software products. There is still a need, however, for better understanding of testability issues and for testing techniques that can operate cost-effectively on SPLs. In this paper we consider these testability issues and highlight some differences between optional versus alternative features. We then provide a graph based testing approach called the FIG Basis Path method that selects products and features for testing based on a feature dependency graph. We conduct a case study on several non-trivial SPLs and show that for these subjects, the FIG Basis Path method is as effective as testing all products, but tests no more than 24% of the products in the SPL.

1 Introduction

Software product line (SPL) engineering has been shown to improve both the efficiency of the software development process and the quality of the software developed, by allowing engineers to systematically build families of products with well defined and managed sets of re-usable assets [4]. A large body of research on SPL engineering has focused on reuse of core program assets [4, 15, 17], refined feature modeling [8, 9, 23], and code generation techniques [2, 7]. There has also been research on testing software product lines [3, 6, 10, 24].

Despite this prior work, there still remains a need to improve reuse during the software testing process. Kolb and Muthig [15] point out that testing has not made the same advances as other parts of the SPL lifecyle and remains a bottleneck in SPL development. Their work highlights issues related to testability of SPLs, where testability is viewed as the ease with which one can incorporate testing into the development cycle and increase reuse while retaining a high rate of fault detection. They comment that the primary strength of SPL development, variability, also has the greatest impact on reducing testability [15], due to the combinatorial explosion of feature combinations that occurs as variability increases [6, 17]. In other related work, Jaring et al. [13] point out that the testability of a product line can be viewed as a function of the binding time of variability, and that providing early binding can increase the ability to test products early. McGregor [17] and Cohen et al. [6] have suggested ways to reduce the combinatorial space by sampling products for testing using combinatorial interaction testing [5]; this work does not address testability issues.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 241–255, 2010.

While all of this work aims at the core problem of software product line testing, none of it specifically considers reuse by examining the feature model and analyzing testability at a finer grain. In this work we consider testing from this perspective. We drill down into the issue of variability and analyze different types of variability, e.g. optional features versus alternative choice features, as they relate to testability. We conjecture that while the alternative choice features have a negative impact on testability by increasing the number of products, optional choice features do not. We then propose a new black box approach for testing software product lines that attends to these issues. We hypothesize that our approach can reduce testing effort while retaining good fault detection in the presence of alternative features.

Our approach, which we call the FIG Basis Path method, translates a feature model into a feature inclusion graph that is, in essence, a feature model dependence structure. We associate all features with sets of test cases and then walk this graph to generate a subset of independent paths (or products) that cover the graph for testing. This is analogous to the basis path approach for testing software applications [26] which finds a "possibly minimal" set of paths to cover all nodes in a program's control flow graph.

We report results of a case study on two software product lines. In both SPLs we can achieve the same fault detection results as we can testing all products. Further analysis shows that we can also use a grouped variant of the Basis Path algorithm to test subfamilies of the SPL as defined by the alternative features.

2 Background and Related Work

There has been a lot of work on feature modeling of which we present only a small subset [1, 2, 14, 19, 21]. In a feature model, a product line can be represented by mandatory and variable features. The variable features may be optional or alternative choices. In their simplest form alternative choice features allow for an exclusive or relationship. We can also have cardinalities assigned that allow for $0...n$ or $1...n$ relationships where the first number is the lower bound and the second number is an upper bound. An exclusive or alternative feature is usually the default for alternative features in a model and is a 1...1 relationship.

We present a small product line to help to explain these ideas and to illustrate our techniques. The feature model for this product line (shown using the Orthogonal Variability Model (OVM)) [19] is seen in Figure 1 on the left (the right portion of this figure will be discussed later). This product line defines a family of 42 calculator programs. It has a core set of features, Exit and Clear and one optional feature, Backspace. Users can select one of three languages (English, Chinese or Spanish) used in the menus, titles and help. Finally the memory features include Memory Store and Recall.

The mandatory features in our sample product line are Core and Language. Within each of these there is some variation. Store is an Or feature from Memory and it is required only if the Memory Recall feature is selected.

Feature models have been used for generative programming [2, 7, 23], providing a model based approach to the realization of product lines. Feature models

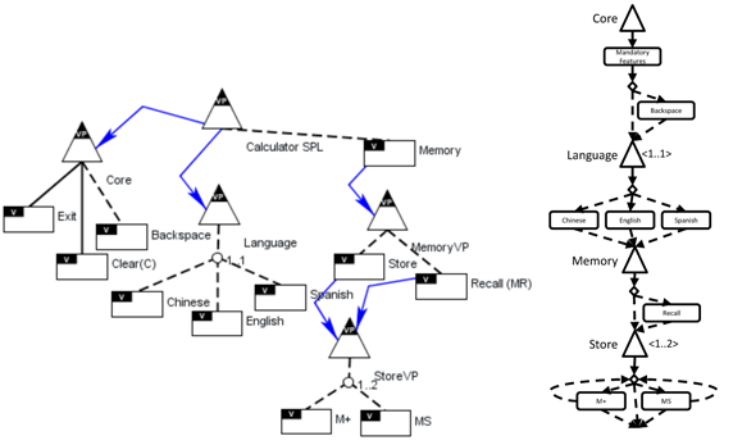


Fig. 1. Calculator SPL Feature Model

have also been used to model the product space for instantiating products for testing [3, 6, 24]; for instance, the work of Uzuncaova et al. [24] transforms the feature model into an Alloy specification and uses this to generate test cases, while the work of Cohen et al. [6] uses the feature model to define samples of products that should be included in testing. Similarly, the PLUTO methodology [3] uses the feature model to develop extended use cases that contain variability which can formulate a full set of test cases for the family of products. Schürr et al. [22] use a classification tree method for testing from the feature model. Other extensions of feature models have been for staged generation [8] or modeling constraints [9]. None of this work explicitly uses the feature model as we do, in a graph based representation, that can be used to select products (and test cases) for testing. The work of Bachmeyer et al. [1] also uses a graph based representation of a feature model, but they do not use this in the testing process.

Other work on software testing product lines includes that of Denger et al. [10] who present an empirical study to evaluate the difficulty of detecting faults in the common versus variable portions of an SPL code base concluding that the types of faults found in these two portions of the code differ. They use both white and black box techniques but do not test from the feature model.

3 Leveraging Redundancy for Testing via Feature Models

We begin with the conjecture that black box testing of software product lines can be made more efficient and effective by designing the product line architecture (and resulting feature model) in a manner that supports reuse of product line testing results across different products. Others have argued that variability decreases testability [15], but we believe that there should be a finer grained examination of this argument. Both optional and alternative features can be viewed as points of variability in a software product line, yet we believe they

may behave differently from a black box testing approach and provide different opportunities to reduce testing effort, as we explain next.

Our methodology involves the following steps: (1) transform the feature model into a *feature inclusion graph*; (2) associate use cases with each feature; (3) develop test cases for each use case; (4) Select basis paths on this graph; and (5) for each path (product), run all test cases for the included features.

3.1 Feature Inclusion Graph

In this section we present a transformation of the feature model into a graph that we call a *feature inclusion graph* (FIG), which represents feature dependencies derived from the feature model. In a FIG, all features that appear on a non-branching path are included in the same product, while branches represent the variability in feature composition. We view the FIG as having a loose connection to the control flow graphs used in software testing; a control flow graph shows explicit flow of control in a program and can be used to select test cases for white box testing. Harrold [12] has suggested that regression testing techniques can be applied to different abstractions of software artifacts as long as they can be represented as a graph and tests can be associated with edges. In our scenario we do not have control flow; rather, our paths represent a combination of features and its dependencies, but we use a common method from control flow based testing to find a *basis path set* [26] for the graph – a set of independent paths through the program graph.

The FIG contains all features of the SPL. We next show how it is derived using different parts of a feature model from OVM [19]. In OVM, the core concepts of an SPL model are the variation points and variants. Each variation point (VP) has at least one variant and the edges between VPs and variants indicate dependencies. In a FIG we apply the same OVM concepts. A FIG has two main components, features and edges. The edges represent the variability of our diagram making explicit all possible paths that we can traverse to generate the minimum set of products. The features are classified as Mandatory, Optional, or Alternative. Next, we describe how each feature and edge are represented in the FIG and how they interact with each other.

In a FIG, a variation point is represented as a triangle and variants are represented as rounded rectangles. A Mandatory dependency is represented by a solid edge between a VP and variant, while an optional dependency uses a dashed edge. A diamond represents the variability of optional and alternative features. Figure 2 shows an example of Mandatory and Optional features represented in the OVM language and FIG diagram, respectively. In this example we see on the left (Figure 2) two mandatory features in OVM (B and C). These are both required in the same *flow of control* therefore we put them on a single non-branching edge of our FIG (lower bottom of figure). Note that either B or C can come first since the dependency is only important at the branches (e.g. this is a partial order). On the right (Figure 2) we show two optional features (again B and C). Here we have added three branched edges. The middle edge represents the case in which neither feature is included, while each of the other

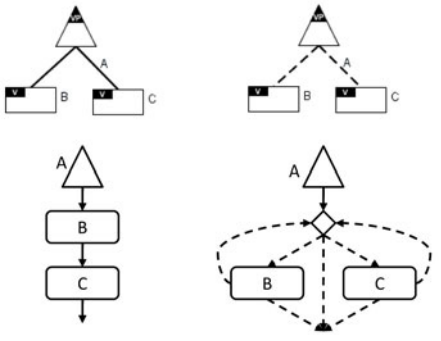


Fig. 2. Mandatory and Optional Features

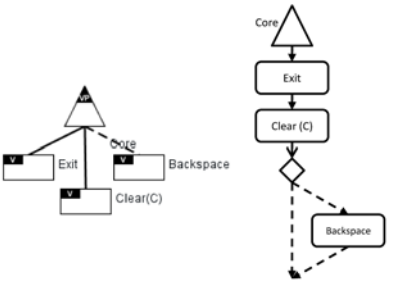


Fig. 3. Calculator Example

edges allow for the inclusion of either feature. We also include a back edge for each feature since it is possible to include a second feature. Assuming that we allow only one instance of a feature for a single product, we can see that there are four possibilities in this graph: we can have no optional features, one of B or C, or both B and C.

From a testability perspective we view this type of variation to be more testable than some other types of variation, since we can include both features (B and C) in a single product. With two optional features we have a 75% reduction in the number of products that we must instantiate in order to test all features. We can apply this to the Core Variation Point and its variants in the calculator SPL. The Core Variation Point has two mandatory features (Exit and Clear (C)) and one optional feature shown in Figure 3. As we can see in this case the optional feature (backspace) can be included in the first product tested providing us with a single instance.

Alternative features are features that are mutually exclusive and present a more difficult challenge for testability. We argue that these are the true deterrents to testability since only one feature can be present in an SPL at a time. Even with these types of variation points we may still gain some benefit in reusability. Figure 4 shows three examples of alternative features in OVM (top)

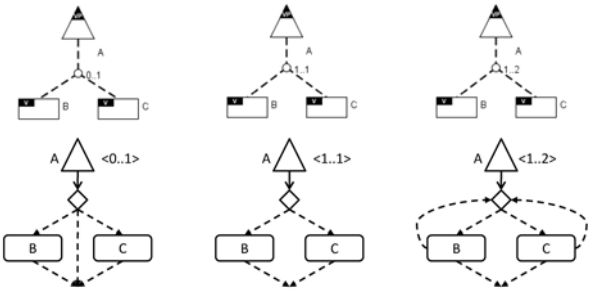


Fig. 4. Alternative Features

and its corresponding FIG (bottom). The first example has cardinality 0...1, i.e., this is really an optional feature and we can include at most one of the two alternatives. In this case we expect to see a small benefit from the optional feature characteristics. We need two of the three possible products to cover all features.

The second example (middle) shows the exclusive or 1...1 relationship. This is the least testable type of variation since it forces the combinatorial space to increase. Here we have two dashed edges to B and C, no back edges and no middle edge. We have two possible products and need to test both to cover the features of this graph. We see no reduction.

The last example (right) is when we have a 1...n relationship; the figure shows a 1...2 relationship. We have a back edge from each feature, and we can cover all features using a single product even though there are three products (B, BC, and C), by including both B and C in our product for testing.

The graphs do not explicitly incorporate constraints in the representation. We maintain a separate set of constraints that we can check during our graph traversal, to ensure consistency, but will examine this in future work.

3.2 Selection Algorithms

In this section we present four methods for selecting products for testing. The first two use the FIG and the second two do not. The first algorithm is our core algorithm called the *FIG Basis Path* algorithm. The idea is to select a set of independent paths in the program that cover all features in the graph. We then present a variant of this called the *FIG Grouped Basis Path* algorithm, that tests subfamilies of the product line grouped by the alternative features in the SPL. We believe that this algorithm will be incorporated into the development process more smoothly, where one particular subfamily is created at a time. The third algorithm does not use the FIG, but is used in our empirical comparison as a method that we believe will be less expensive; we call this the *All Features* algorithm. This algorithm greedily chooses products until all features in the product line have been included at least once. The last method we discuss is also used as a basis for comparison. We expect that it will be stronger than the other comparison method, but also perhaps more expensive. This is the *Covering Array* method suggested by McGregor [17] and Cohen et al. [6]. In this method we select a subset of products from the feature model that cover all pairs of features in the SPL. We describe each method in more detail next.

The **FIG Basis Path Algorithm** (Algorithm 1) is based on the basis path algorithms in [26]. In this algorithm we assume that the FIG is built and that we have a set of constraints on the features. We begin by setting the basis path set (BP) to be empty. We then iterate through all paths in the FIG in a depth first traversal order (to ensure we find the longest paths first).In the algorithm we reference the authors use a breadth first search, but our objective is slightly different. For each path we check the constraint set to see if the path is feasible. If it is, we then check if it is linearly independent with the other paths in BP. (In our study we performed this step manually, however, it can be automated with a constraint solver.) If it is independent we add it to BP. For example,

Algorithm 1. FIG Basis Path Algorithm

```
BasisPath(FIG)
BP = ∅
for all paths, P, ∈ FIG (using DFS order) do
   if P is feasible then
      if LinearlyIndependent(P, BP) then
         add P to BP
      end if
   end if
end for
```

suppose we want to select the minimum set of products in the calculator SPL. We show the FIG on the right portion of Figure 1. For each path, we evaluate if it is feasible by checking existing constraints. In this case, all paths that include the Memory Recall variant and do not include the Store variant will be removed from the final set of paths (Products). We next begin our selection. In the first path, 6 variants are selected, containing all of the mandatory features (Exit, and Clear (C)) and optional features (Backspace), one language - Spanish - and two variants from the Memory variant point: Store (M+) and Recall. The second path substitutes the Store variant from the previous path (M+) to MS. The third and fourth paths change only the Language variant.

The **FIG Grouped Basis Path Algorithm** is a modification of the FIG Basis Path algorithm, in which subfamilies of the product line are grouped based on the alternative features. We begin by generating all of the paths in the FIG in depth first order and check each for feasibility. We then group all feasible paths by alternative feature groups, where all paths that include a particular alternative feature are included in its group. If there are paths that contain no alternative features, we create an additional group to hold these. For example, suppose we want to group based on the language VP in the calculator SPL. In this case we would find all paths that contain Spanish and put them into one group. All that contain Chinese go into in another, and the rest that contain English are put into another. Once we have the grouping, we use the Basis Path algorithm for each group where the FIG is replaced with the set of paths in the group. We can skip checking feasibility since this has already been performed.

Our third algorithm, the **All Features Algorithm**, does not require a FIG. This algorithm is less expensive than the first two because it does not involve enumerating paths and walking the graph. Instead its goal is to include a set of products that just cover all features. We begin by placing all features into one of two sets, Mandatory and all others. We include additional constraints to enforce our alternative features, then we order features in descending order based on the the number of constraints on that feature. We keep a set we call *used features* which starts as empty. For each product, we greedily add features (putting them in the constraint order described) into a product, skipping those that are already in the used feature set (unless mandatory or part of a requires constraint), or that violate a constraint, until we have a product including the greatest number of unused features. We then update the used features set. Once all features have been included in at least one product we are done. For the calculator SPL,

we create 3 products. The first product contains the mandatory features, the Chinese language and all variants associated with the Memory variant. The second and third products do not include any variant from Memory, but the Language variant is changed to English and Spanish, respectively.

Our fourth algorithm, the **Covering Array Method** uses a pair-wise approach and covers all pairs of features in at least one product. This technique tests interactions between features and has been shown to be effective in testing many types of configurable software [20]. The base object used to select the sample is a covering array. A few differences can be noted between this and our other methods. First, in the Covering Array method we consider optional features as being both included and not included. Therefore we would not be able to simply test a product with both A and B but would need to test A with and without B, and B with and without A as well as neither feature. While possibly a stronger testing criterion we expect that this method will be more expensive and may not be helped by improved testability as we have described it.

4 Case Study

To gain insights into the operation of the FIG basis path approach we conducted a case study, comparing the approach to the three other approaches described in Section 3. Our goal is to address the following research questions:

RQ1: How does the FIG basis method compare with other test methods?
RQ2: Can we reduce the effort required to test groups of products through the grouped basis method?

4.1 Study Objects

As objects of study we selected two software product lines, both developed by other researchers and used in previous studies of SPLs. The first SPL is a Graph Product Line (GPL) created by Lopez-Herrejon and Batory [16]; it is built using the AHEAD methodology and implemented as a series of .jak files [2] (an extension of the Java language). The second SPL is a Mobile Software Product Line [11] created by Lancaster University and widely used in previous studies.

Table 1 lists, for each of our software product lines, the total number of lines of code excluding comments (*LOCs*), the number of classes present (*Classes*), the number of products that can be created (*Products*), the number of faults present (*Faults*), the number of variants classified as Optional, Alternative, and Or (*Variants*) and the number of constraints classified as Require and Exclude (*Constraints*). The total number of lines of code (LOCs) corresponds to the product that has the most features selected.

The Graph Software Product Line (GPL) is an SPL that implements a family of graph algorithms in which each product is a type of graph. The code base includes 1435 lines of jak code and consists of 15 features. A graph is either directed or undirected. Edges can be weighted or unweighted. A graph product

Table 1. Objects of Study

					Variants			Constraints	
	LOCs	Classes	Products	Faults	Opt.	Alt.	Or	Require	Exclude
GPL	1435 (jak)	12	38	60	0	4	1	10	1
MobileMedia - V5	2220	37	16	10	4	0	0	0	0
MobileMedia - V6	2173	38	24	10	4	0	1	4	0

requires at most one search algorithm: depth-first search (DFS) or breath-first search (BFS), and at most one or more of the following algorithms: vertex numbering, connected components, strongly connected components, cycle checking, minimum spanning tree and single-source shortest path. The GPL feature model contains a total of 80 instances without constraints. After reading the documentation for the GPL we created a feature model for it, as shown in Figure 5. To create this model we needed to resolve some ambiguity in the documentation and we also reduced the possible cardinality for combinations for the variant point Alg. Ultimately we obtained 38 possible instances of the product.

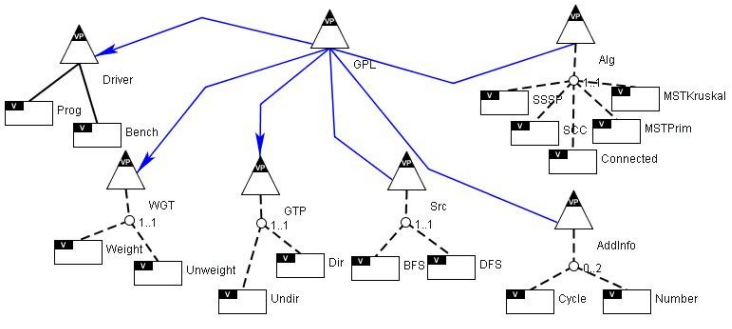


Fig. 5. Graph SPL Feature Model

Mobile Media is an SPL that implements mobile applications that manipulate media (photos, music and video) on mobile devices. Mobile Media has evolved since 2005 to support several types of media. Mobile Media has nine releases implemented in two paradigms: aspect oriented and object oriented.

For our study, we selected two versions of Mobile Media that were developed using the object oriented paradigm, versions 5 and 6. In version 5, users are allowed to manipulate image files in different mobile devices as well as send messages, set favorite pictures, copy images and perform other operations. Version 6 is a refactored version of version 5 and it includes one more variation point. In this version, users are allowed to manipulate two different types of media: photo and music. Both versions share a few operations but they have different underlying code bases due to the refactoring. The Mobile Media Feature Model allows us to derive a total of 16 and 24 instances for version 5 and 6, respectively. Figure 6 presents the feature model for Mobile Media and its evolution.

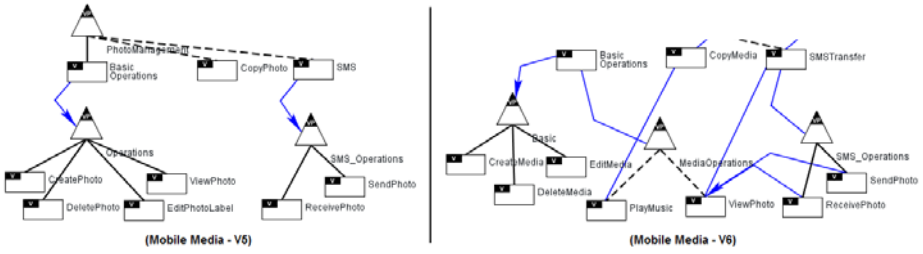


Fig. 6. Mobile Media SPL Feature Model

4.2 Test Suites

To conduct our study test suites were developed by associating each feature with its correlated use case requirements. For each feature, we developed concrete test cases that cover the primary scenario as well the alternatives use cases. We used the documentation provided with the object to generate these. All test cases were created by other researchers in our group not including the authors of this paper. For the GPL product line, the test suites are command line test cases. For Mobile Media the test cases are GUI based and implemented using a combination of two open source testing tools, Microemulator [18] and Abbot [25].

4.3 Fault Seeding

In the Mobile Media application, during the course of working with the system, we found 10 actual faults that caused the system to working improperly.We corrected each fault based on the requirements provided with the application and then re-seeded each fault into a single *faulty version*. We thus had 10 faulty versions of this application for both version 5 and 6.

For the GPL application, we provided six students in our laboratory, who were not involved with the study itself and had no knowledge of the approaches being studied, with a document on an approach for doing fault seeding and subsets of the .jak files. We asked each student to seed 10 faults into their set of files. This yielded 60 faulty versions of this application.

4.4 Study Conduct

To conduct our study we applied each of the four testing approaches to each of our fault free objects. We executed these test cases on our faulty versions, and to determine whether a test case detected a fault, we compared the output of the faulty version under that test with the output of the original (non-faulty) version of the object under that test. All of our executions were performed on a 1.8GHz Intel Pentium M with 1GB of system memory running SuSE Linux 10.1 platform equipped with the Java 1.5 JDK.

5 Results

In this section we examine the results of our research questions. We begin with RQ1 which asks how the FIG Basis Path method compares with other methods. Table 2 shows the data for both applications. The first column shows the number of products tested, followed by the number of test cases run. The rightmost column shows the number of faults detected by each technique.

In considering this research question we examine three methods: the Covering Array method, the All Features method, and the Basis Path method, and we compare these to an All Products method which performs an exhaustive enumeration of all products. (The Grouped Basis Path method is considered for RQ2.) As the table shows, in the GPL of the 60 faults inserted, 54 were found when we tested all products. Both the Covering Array method and the Basis Path method also found 54 faults, however the Basis Path method used fewer than half as many products as the Covering Array method and 39.7% of the test cases. The least expensive method was All Features (5 products and 26 test cases); however, this technique missed 3 faults when compared with the other techniques.

We next consider the results for the two versions of Mobile Media. In this case we see that all methods found all of the faults in both versions. For version 5, the All Features and Basis Path methods required only one product and 49 test cases, compared with 348 test cases for the All Products method and 190 test cases for the Covering Array method. For version 6, the All Features and Basis Path methods required only two products with 71 and 85 test cases respectively, compared with 839 test cases for the All Products method and 201 test cases for the Covering Array method. We discuss the implications of these results in the next section.

To answer RQ2 we examine data shown in Table 3. This table shows the data grouped by the alternative features of each SPL. The left side of the table

Table 2. Number of Test Cases and Faults Detected by Technique: Mobile Media SPL

	Graph SPL Total Number of Products: 38 Total Number of Faults: 60		
Method	**# Products**	**Test Cases**	**Faults Detected**
Covering Array	20	141	54
All Features	5	26	51
Basis Path	9	56	54
All Products	38	256	54

	Mobile Media 5 Total Number of Products: 16 Total Number of Faults: 10			Mobile Media 6 Total Number of Products: 24 Total Number of Faults: 10		
Method	**# Products**	**# Test Cases**	**#Faults Detected**	**# Products**	**#Test Cases**	**# Faults Detected**
Covering Array	5	190	7	6	201	10
All Features	1	49	7	2	71	10
Basis Path	1	49	7	2	85	10
All Products	16	348	7	24	839	10

Table 3. Number of Test Cases and Faults Detected by Alternative Variants

Graph SPL						
	All Feasible Paths			Grouped Basis Path		
Variant	#Product	#TC	#Faults	#Product	#TC	#Faults
Shortest	3	19	27	2	13	27
SCC	4	34	28	2	17	28
CC	6	46	21	2	17	21
MSTP	3	14	26	2	11	26
MSTK	3	29	29	2	24	29
Mobile Media v6						
	All Feasible Paths			Grouped Basis Path		
Variant	#Product	#TC	#Faults	#Product	#TC	#Faults
Music	4	139	10	1	40	10
Photo	5	220	10	1	49	10

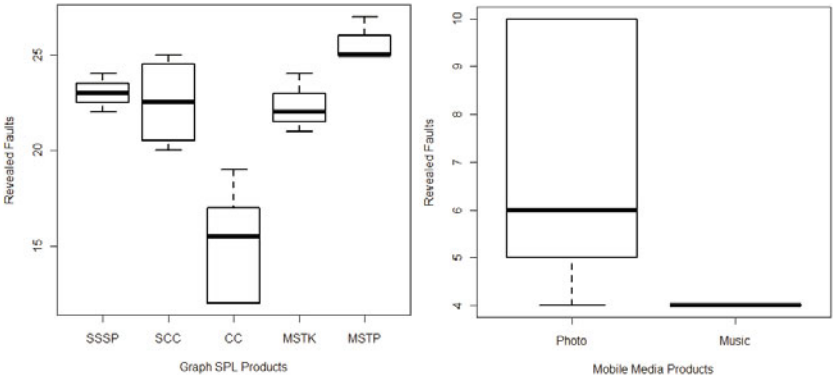


Fig. 7. Number of Test Cases and Faults Detected Grouped by Alternative Features

shows data for all feasible paths in each group, including the number of products, number of test cases, and number of faults detected. The right side of the table shows the same data for the selected products using the Basis Path method. In every group of products we see that we can reduce the number of products tested while retaining the fault detection capability. For GPL, our reduction in products ranges from 67% (CC) to 33% (Shortest, MTSP, MSTK). In addition we have used between 37% and 79% of the test cases required for all feasible paths, resulting in at least a 20% reduction in the required test cases. For Mobile Media, we had a reduction of between 71.2% and 77.7% of the test cases and 75% (Music) to 80% (Photo) of the products.

In an additional analysis we wanted to determine whether any subset of n paths could have been selected with the same fault detection results within each group. For GPL, we performed a pair wise comparison between products since we have selected two products for each group. For each group we combined all combinations of 2-paths and calculated the fault detection. We show this data in the form of a box plot (Figure 7). In each plot we see a range of fault detection, indicating that the Basis Path method is providing the best fault detection (we

know that it is at the top of the box plot since all basis path results provided the same fault detection as the full set of feasible paths in that group). Since the Grouped Basis Path for Mobile Media selected only one product from the whole set of feasible products, we evaluated the fault detection for all products that belongs to the same group. The data in Figure 7 shows that there is a range of fault detection between products in the Photo variant, but products with the Music variant selected have the same fault detection. This confirms that we cannot randomly select a subset of products within groups and necessarily be assured of the same fault detection.

5.1 Discussion

For RQ1, based on this data we believe that the FIG Basis Path method is efficient at finding faults and is at least as effective as other techniques. In the product line that we define as less testable due to the alternative features (GPL) we see that the Basis Path method performed the best. It found as many faults as the other techniques for 60% fewer test cases than the CA technique, and 55% fewer products. In the Mobile Media application, where we believe we have a more testable product due to the small number of alternative features, we see that the FIG Basis Path was as effective at finding faults as all other methods, and costs the same as the least expensive method, All Features. It was less expensive than the covering array method as well. Given these results we suggest that although the cost of computing the FIG Basis Path may be slightly higher than that for All Features, the technique appears to work well for both types of feature model elements (alternative and optional), therefore it is the more robust technique. Further evaluation is needed to understand the efficiency of using a FIG with higher granularity variants. We believe there is a correlation between the granularity in the feature models and the efficiency of our technique. We also have analyzed where the faults lie within our applications and many faults were located in the mandatory and optional features. We need to further analyze the impact of faults that are embedded inside of the variant portions of the code to fully understand the effectiveness of these techniques.

For RQ2, we see that it is possible to test parts of the product space more efficiently using the Grouped Basis Path method when the feature model has alternative variant points. In the GPL application, where we believe we have a less testable product due to the variability and a large number of requirements constraints, we were able to select a small set of products that revealed all our faults with fewer test cases and products. Furthermore, the boxplots tell us that we cannot simply select the paths to test randomly. This suggests a further use of the FIG Basis Path method, where we want to focus on parts of the SPL at a time or where development is taking place in stages, based on specific variation points. Conversely, the Grouped Basis Path method did not show any improvement over the FIG Basis Path method in the Mobile Media application. We believe the small number of constraints of the Mobile Media application has some influence over the method. We conclude for RQ2 that we can use FIG Grouped Basis Path to reduce test effort.

6 Conclusions and Future Work

In this paper we have used the feature model to drive test case selection and have asked if we can reduce test effort while retaining fault detection capability through a graph-based selection algorithm. Using the FIG Basis Path method we were able to detect the same number of faults as we did when testing all products, by testing as few as 6% and no more than 24% of products in our SPLs, and running only 10% of the test cases as All Products in the best case. The most effective non-graph technique, the Covering Array method, required us to test between 13% and 54% of products respectively in the same systems. In the subject with only optional features, we see that our method does as well as all other techniques in fault detection and costs no more than the least expensive technique, All Features.

In future work we plan to examine this technique on larger software product lines with more complex faults. We will also examine other variations of the feature model such as 1...n relationships and the impact of constraints on our model.

Acknowledgments

We thank B. Gavin, T. Yu, S. Huang, A. Sung, S. Kuttal and W. Xu for help in seeding faults and W. Motycka for developing the test suite for the subjects used in this work. We also thank Eduardo Figueiredo for providing us the source of Mobile Media Software Product Line. This work was supported in part by NSF under grants CCF-0747009 and CNS-0454203, and by the AFOSR through award FA9550-09-1-0129.

References

1. Bachmeyer, R.C., Delugach, H.S.: A conceptual graph approach to feature modeling. In: Intl. Conference on Conceptual Structures, pp. 179–191 (2007)
2. Batory, D.: Scaling step-wise refinement. IEEE Transactions on Software Engineering 30(6), 355–371 (2004)
3. Bertolino, A., Fantechi, A., Gnesi, S., Lami, G.: Product line use cases: Scenario-based specification and testing of requirements. LNCS, pp. 425–445. Springer, Heidelberg (2006)
4. Clements, P., Northrop, L.M.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading (2001)
5. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: an approach to testing based on combinatorial design. IEEE Transactions on Software Engineering 23(7), 437–444 (1997)
6. Cohen, M.B., Dwyer, M.B., Shi, J.: Coverage and adequacy in software product line testing. In: Workshop on the Role of Architecture for Testing and Analysis, pp. 53–63 (July 2006)
7. Czarnecki, K.: Overview of generative software development. In: Banâtre, J.-P., Fradet, P., Giavitto, J.-L., Michel, O. (eds.) UPP 2004. LNCS, vol. 3566, pp. 313–328. Springer, Heidelberg (2005)

8. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multilevel configuration of feature models. In: Software Process: Improvement and Practice, pp. 143–169 (2005)
9. Czarnecki, K., She, S., Wasowski, A.: Sample spaces and feature models: There and back again. In: Intl. Software Product Line Conference, pp. 22–31 (2008)
10. Denger, C., Kolb, R.: Testing and inspecting reusable product line components: First empirical results. In: Intl. Symposium on Empirical Software Engineering, pp. 184–193 (2006)
11. Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., Dantas, F.: Evolving software product lines with aspects: an empirical study on design stability. In: Intl. Conference on Software Engineering, pp. 261–270 (2008)
12. Harrold, M.J.: Architecture-based regression testing of evolving systems. In: Workshop on the Role of Architecture for Testing and Analysis, pp. 73–77 (July 1998)
13. Jaring, M., Krikhaar, R.L., Bosch, J.: Modeling variability and testability interaction in software product line engineering. In: Intl. Conference on Composition-Based Software Systems, pp. 120–129 (2008)
14. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute (November 1990)
15. Kolb, R., Muthig, D.: Making testing product lines more efficient by improving the testability of product line architectures. In: Workshop on Role of Software Architecture for Testing and Analysis, pp. 22–27. ACM, New York (2006)
16. Lopez-Herrejon, R.E., Batory, D.S.: A standard problem for evaluating product-line methodologies. In: Bosch, J. (ed.) GCSE 2001. LNCS, vol. 2186, pp. 10–24. Springer, Heidelberg (2001)
17. McGregor, J.D.: Testing a software product line (cmu/sei-2001-tr-022). Technical report, Carnegie Mellon Software Engineering Institute (2001)
18. MicroEmulator (2010), http://www.microemu.org/
19. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering. Springer, Berlin (2005)
20. Qu, X., Cohen, M.B., Rothermel, G.: Configuration-aware regression testing: An empirical study of sampling and prioritization. In: International Symposium on Software Testing and Analysis, pp. 75–85 (July 2008)
21. Schobbens, P.-Y., Heymans, P., Trigaux, J.-C.: Feature diagrams: A survey and a formal semantics. In: Intl. Requirements Engineering Conference, pp. 136–145 (2006)
22. Schürr, A., Oster, S., Markert, F.: Model-driven software product line testing: An integrated approach. In: Theory and Practice of Computer Science, pp. 112–131 (2010)
23. Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe composition of product lines. In: Intl. Conference on Generative Programming and Component Engineering, pp. 95–104 (2007)
24. Uzuncaova, E., Garcia, D., Khurshid, S., Batory, D.: Testing software product lines using incremental test generation. In: Intl. Symposium on Software Reliability Engineering, pp. 249–258 (2008)
25. Wall, T.: Abbot Java GUI test framework (2010), http://abbot.sourceforge.net/doc/overview.shtml
26. Yan, J., Zhang, J.: An efficient method to generate feasible paths for basis path testing. Information Processing Letters 107(3-4), 87–92 (2008)

Architecture-Based Unit Testing of the Flight Software Product Line

Dharmalingam Ganesan[1], Mikael Lindvall[1], David McComas[2], Maureen Bartholomew[2], Steve Slegel[2], and Barbara Medina[2]

[1] Fraunhofer Center for Experimental Software Engineering, 20740 College Park, Maryland, USA
{dganesan,mlindvall}@fc-md.umd.edu
[2] NASA Goddard Space Flight Center (GSFC), 20771 Greenbelt, Maryland, USA
{david.c.mccomas,maureen.o.bartholomew,steve.slegel, barbara.b.medina}@nasa.gov

Abstract. This paper presents an analysis of the unit testing approach developed and used by the Core Flight Software (CFS) product line team at the NASA GSFC. The goal of the analysis is to understand, review, and recommend strategies for improving the existing unit testing infrastructure as well as to capture lessons learned and best practices that can be used by other product line teams for their unit testing. The CFS unit testing framework is designed and implemented as a set of variation points, and thus testing support is built into the product line architecture. The analysis found that the CFS unit testing approach has many practical and good solutions that are worth considering when deciding how to design the testing architecture for a product line, which are documented in this paper along with some suggested improvements.

Keywords: unit testing, implemented architecture, mock, function hook, coverage, flight software.

1 Introduction

It is a well-known fact that the cost of finding and fixing a bug at the time of unit testing is cheaper than finding and fixing bugs that are found during integration testing, system testing or in the field. In addition, unit tests help developers while performing software changes because they indicate when changes break existing functionality. However, unit testing is not easy in practice for reasons including a) modules often depend on other modules, making them hard to separate and unit test in an independent fashion, and b) modules can also depend on unique features and functions provided by the operating systems, and they may require the hardware in-the-loop for the software to function properly, making it difficult to set up a controlled unit test environment. In the context of software product lines, one of the important concerns is the capability to unit test core modules without running and being dependent on the behavior of any other core modules, which might not be developed or correct at all times and for all possible scenarios. This capability is important because

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 256–270, 2010.

the whole point of unit testing is to test an individual unit and to produce early and quick feedback regarding the test results.

In the context of product lines the situation is even more complex. For example, the core team must demonstrate the quality of their unit tests to the application team in order to build confidence regarding the quality of the core modules. Furthermore, when the application teams configure the variation points (e.g., features and modules to enable) of core modules or when they modify the source code of core modules, they need unit tests to help them quickly validate the correctness of the software. Because flight software is mission critical and needs to be of very high quality, the flight software branch at NASA GSFC has developed a practical approach for unit testing of its flight software product line (CFS). The Lunar Renaissance Orbit (LRO) mission which is currently orbiting the moon is one successful example usage of the cFE (core Flight Executive), which is the core of the CFS.

This paper discloses the architecture of the unit tests that are used in CFS, with the hope that other product line organizations may benefit from these ideas and concepts. The unit testing strategies described in this paper are sufficiently general and therefore also applicable to other product lines. The central ideas of the unit test architecture provided here include the ability to manipulate return codes of functions that are defined in dependent modules, used by the function under test. In fact, the CFS unit testing framework is designed and implemented as yet another set of variation points, and therefore testing is built into the product line architecture. Thus, the architecture supports plug-and-play of modules where modules can be bound to stub modules for testing, and each instance of a product line can be assembly of mock modules. This supports incremental unit and integration testing because when it has been determined that a certain application works as expected using the stubbed modules, the "real" modules can incrementally be added, one by one, and the same unit tests can be executed again with growing confidence in the final product variant.

The results of the analysis of the CFS unit testing strategy and collection of unit tests show that they share a common look-and-feel in terms of the way they set-up the tests, manipulate return codes of functions defined in other modules they use as well as how they set-up makefiles to run the unit tests. Furthermore, the dependent modules need not compile or run, thus this strategy provides early and quick feedback on unit test results of the module under test.

The analysis of test coverage shows that all publicly visible APIs have dedicated test programs, and many of the internal functions are indirectly tested through the test programs developed for public APIs. Thus, all functions of each core module can be unit tested automatically. For each configuration parameter, there is a dedicated set of unit tests that test the behavior of the relevant functions with respect to the boundaries of the values of individual variation points. The analysis also identified a few design problems from the unit testing point of view. One of the problems is that some functions return the same return code from different paths, making it difficult to determine whether or not the given test input data traversed the intended path of the function under test. This example demonstrates the importance of design for unit testing. Another problem is that some of the unit tests are lengthy due to the fact that they try to test more than one scenario inside one test function, making it difficult to trace back from test failures to the exact scenarios that failed. These problems are already added to the CFS issue tracking system and are being addressed by the CFS team. An important premise of this

statement is that the unit tests are consider an integral part of the product, and configuration is managed just like the source code.

Contributions of the paper. While the software product line community has a growing collection of articles related to modeling and managing variability, there are only a few practically inspired and validated technical papers focusing on unit testing in the context of software product lines. To this end, we hope this paper makes the following contributions:

1. A practical method for unit testing in the context of product lines. The method is derived from the way the CFS team implemented unit testing and examples for the CFS are used to explain the method.
2. A simple, yet effective approach for extracting and analyzing the architecture of the unit tests including a list of criteria were used for reviewing the unit tests and which can be used by other analysis teams.
3. An improved understanding of the relationship between software architectural design and unit tests. That is, an understanding of what makes unit testing easier or harder to develop and maintain. In addition, the paper demonstrates, using concrete examples, the importance of following architectural rules to facilitate unit testing.

It should be noted that while this paper focuses on unit testing, other important types of testing such as integration and system testing are also needed, and are briefly discussed at the end of the paper.

2 The CFS Product Line Architecture

This section introduces the CFS product line architecture as a context for understanding the architecture of unit tests. For CFS business goals and heritage, see [2]. The CFS has a layered structure. The top level layer has a catalog of reusable mission independent modules (a.k.a. applications), which may be used in one or more missions. Mission-specific modules (a.k.a. applications), i.e. they are only used in one mission, are also part of this top level layer. The second level layer (the Core Flight Executive (cFE) services layer) is the core of the CFS. The core layer offers several services, for example, the software bus module for inter-application communication, and the executive service module that manages the lifecycle of each application on the top level. Below the core layer, there is an OS abstraction layer (OSAL) which offers a common API for all operating systems supported by CFS (e.g. Vxworks, Rtems, and Unix), which was also released as open source [4]. There is also a board support package layer (BSP) which loads the configured OS and boots the CFS as well as a hardware abstraction layer, which offers a hardware-independent API for different types of hardware processors and ports, see Fig. 1. The cFE services and its lower layers are offered to various missions both inside and outside the NASA including a catalog of CFS applications that can be reused. Each cFE core service is configurable by choosing the values for appropriate constants declared in the interface or header files of each service.

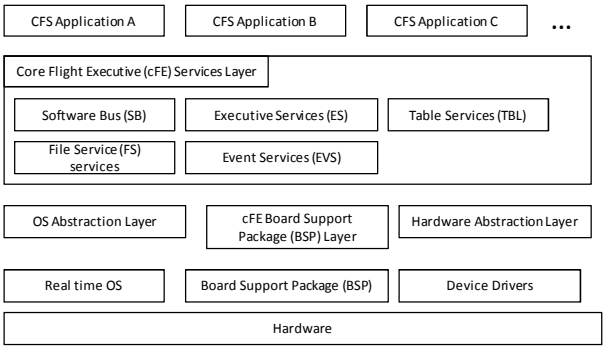


Fig. 1. The structure of the CFS Product Line

All CFS modules are fully implemented in the programming language C. Each module has a set of C files with configuration parameters and public API functions declared in header files. There are dedicated makefiles for each module, which compiles all its files and produces an object file. All core modules are linked into one shared core library. Missions reuse this shared library and develop applications using the APIs offered by the core modules. Missions can add their own application modules to the top level application layer. However, in order to preserve the built-in flexibility and run-time reconfigurability, applications do not communicate directly with each other. Instead, applications communicate by subscribing to and publishing messages from the software bus and it is the responsibility of the software bus to deliver messages to all subscribed applications, see Fig. 2. The software bus is an abstraction built on top of OS queues and sockets making the applications unaware of the communication mechanism, which thus can be chosen at build time.

In [2], the CFS source code was analyzed with respect to its compliance to architectural rules. The detected violations of the architectural rules have now been removed and a new version of the CFS has been released. The previous analysis concluded that the CFS implementation is indeed consistent with the specified architecture. That is, layering is in place, and all CFS applications communicate only using the software bus. In this paper, we focus on the CFS' unit testing strategy and will explain the architecture of unit tests based on the software architecture of the CFS. The overall high level question is how we can test CFS-like product lines, which have to be of very high quality. The first step towards such testing is unit testing.

3 Technical Set-Up and Process for Reviewing Unit Tests

This section introduces the approach followed for the independent review of unit testing strategies and their accompanying unit tests etc. For this paper, the review was applied in an independent way: the CFS team provided the artifacts to the Fraunhofer team, which has not been involved in any way with the development or testing of CFS, for review, feedback and recommendations for improvements. The Fraunhofer team used their reverse engineering and software architecture competency to review the unit tests of the CFS. It was the task of the Fraunhofer team to independently

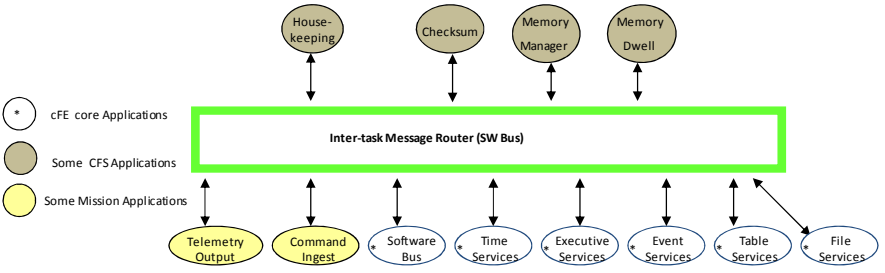


Fig. 2. The context diagram of the software bus in the CFS. Each module (a bubble) runs in a separate task, and communicates with other modules by publishing and subscribing to messages using the software bus.

understand how the unit testing is performed, and to identify issues with the current practice. These issues were then presented to the CFS team in technical meetings with the CFS engineers, project leaders, test leaders, etc. This process has been going on for 2 years and is reiterated whenever new releases or test suites have been developed.

The main goal of the review process is to get a good overview of the current state of the test suites. In order to do so, the review process attempts to formulate answers to the questions listed in Table 1. The approach to answering the questions is based on first extracting the architecture of unit tests from the test code. After that, the extracted unit test architecture was analyzed to understand the strengths and weakness. The practically relevant questions listed here are answered using a semi-automatic approach based on a reverse engineering and visualization tool suite used by the Fraunhofer team in projects with customers.

The Data Extraction Step: This first step involves parsing the existing source code and test suites to extract relations between entities (e.g. call relations, include relations) at the code level. The extracted relations are stored in a relational format in two databases: one database stores source code relations, and the other database stores relations of the test suite. This extraction is completely automated using parsers developed at Fraunhofer. The makefiles are also needed for the analysis because a) they contain information related to compiler switches, preprocessor symbols, and header files b) they contain information related to which object file is linked with the other object files. This linking knowledge is vital to extract correct dependency diagrams among modules, among test suites, and from test suites to source code modules. The ifnames tool is used to extract all conditional preprocessor symbols (excluding header file guards), which are basically variation points supported by the system. These variation points are later used to analyze how test suites handle them.

The Analysis, Query, and Visualization Step: In this step, the extracted data is analyzed using SQL-like queries written based on the RPA toolkit [1]. The RPA language supports several relation and set theoretic operators to query the extracted data. For example, it is possible to extract all functions defined in the source code which are not referenced by any of the test suites. Several RPA queries were developed for answering questions such as: 1) Is the given function tested at all by a test program or is the given function tested indirectly by a test program? 2) Is there is a stub or mock

Table 1. Questions for reviewing the existing unit test code

Question	Purpose
1. Can core modules be tested independently of the core modules it uses?	To a) understand whether modules have unit tests, b) if there are architectural design issues that make unit testing hard.
2. How are variation points of each module being handled during unit testing?	To understand how to unit test the behavior with respect to each variation point (e.g. maximum number of messages in the software bus).
3. How easy is it to create mock or stub implementations of dependent modules?	To understand how complex it is to set-up so-called mock or stub implementations of dependent modules. Ideally, mock implementations are simple and their return values are easy to manipulate to traverse all paths.
4. Can modules be unit tested without access to special hardware and/or OS?	To understand whether modules can be unit tested on standard desktop applications without requiring developers to access special hardware and real-time operating system.
5. How easy it is to set-up the unit tests of a module?	To understand whether it is easy to set-up unit tests. Ideally, with a couple of instructions it should be possible to unit test a function.
6. Are there dedicated test programs for each public function of a module?	To understand, from the coverage point of view: are there unit test programs developed to test each individual publicly visible API. From a reuse point of view, the trust increases if there are dedicated test programs for each API.
7. How lengthy and complex is each test program?	To understand the complexity of unit tests. Ideally, unit tests focuses just on one scenario and do not mix multiple scenarios into one test program. Measuring the length and the number of conditional statements of each test program shed light on how well unit tests are structured internally.
8. How are the test results collected and reported for further analysis?	To understand whether developers or testers can easily track back from test failures to the exact scenario. Are the code coverage results collected and stored either for further investigation or to derive new test cases.
9. Is there a common look-and-feel in terms of the way the modules are unit tested?	To understand whether there is a well-defined architecture for unit tests, including constraints or rules for setting-up mocks, makefiles, set-up of tests and reporting of test execution results. Common look-and-feel a) helps programmers or testers to easily develop new unit tests, b) facilitates understanding of unit tests developed by different developers, and c) improves maintainability of unit test programs.

implementation of this function? 3) How many test programs call this given function, and 4) Which test cases refer the given variation point. While querying is useful to extract information, visualization is very powerful in revealing patterns in the structure of unit tests. Module level dependency diagrams and dependencies of test suites to source code modules were extracted using RPA and visualized using the SAVE tool, whereas the call graph of test suites are visualized using the Prefuse toolkit. The module dependency diagrams were used to review the structure of the test suites in terms of how dependencies to other modules are mocked.

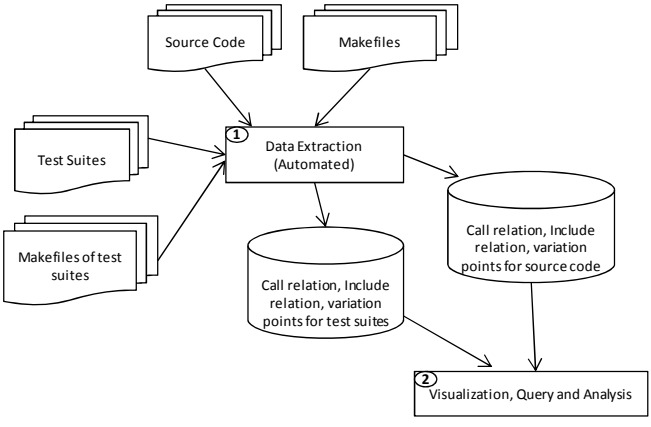


Fig. 3. Two major steps in the analysis of unit test architecture

4 Unit Testing of Core Modules

This section discusses the extracted architecture of the CFS unit tests based on the tool-supported process introduced in the previous section, see Fig. 4. Although it is almost a complete graph, only the offered public interfaces are used and no internal details of modules are shared with other modules. Also, there are clear reasons for each dependency. For example, the Executive Services (ES) module is responsible for initializing all modules, and all modules use the ES to register, create new tasks, or exit their execution. Similarly, all modules use the Software Bus to send and receive messages. The Event Service (ES) module helps modules log important events, and thus it is used by all modules. The File Service (FS) module helps modules write and read file data. The Timing Service module provides timing services to all modules. The Table Service (TS) module helps application layer modules register data tables to share data with other modules. The dependencies were extensively reviewed by the CFS team members and all were deemed valid and necessary. The CFS' Interface Control Document (ICD), which is provided to application teams, specifies the interfaces of each core module. This English specification explains the behavior of each publicly visible function in terms of constraints on the input and the output.

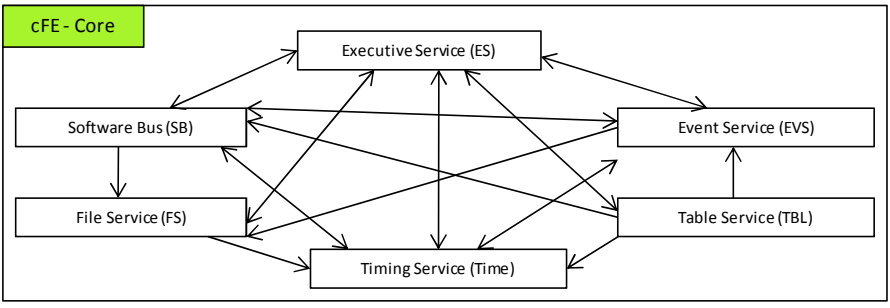


Fig. 4. Dependencies between core modules, extracted from the source code. Arrows represent code relations such as include, call, access of data structures, etc.

Variability in the Core Layer: As described in [2], there are only a few conditional preprocessor statements (e.g. #ifdef, #ifndef statements) in the core layer. For example, the timing service controls some variation points using #ifdefs, such as the Mission Elapsed Time (MET) and Greenwich Mean Time (GMT) formats. There are variation points in all modules, but instead of using #ifdefs, they are declared in the header files of the modules and can thus be configured individually for each mission. These variation points are basically constants. For example, the software bus has a variation point: maximum number of messages in the bus. Even though there are no #ifdefs, the cFE core can be executed on variety OS and hardware architectures because all modules of the core are programmed to the abstract interfaces of OS, hardware, and board support package abstraction layers. Given this brief overview of the architecture of the core layer and how variability is managed at the code level, the remaining section focuses on how each module is unit tested independent of other modules it uses. Fig. 5 shows an example of the high level unit test structure. This example view for the Executive Services (ES) module shows that the stub concepts are used in unit testing. For each core module, such a view was extracted from the test source code. It shows that the ES module depends on stub implementations of interfaces of the EVS, the SB, the Time Services, the Table Services, the File Services module, and also the stubs of OS and board support package APIs. This view is consistent with the source code dependencies of the ES module, shown in the previous Fig. 4, in that instead of using the real implementations of dependent modules, corresponding stubs are used. Note that stubs implement exactly the same interfaces that are implemented by real modules, and stubs are orthogonal – that is they are independent and don't need each other. At link time, the makefile of the module under test links its object files with the object files of stubs it uses. All stubs run in the same thread with the test suite. This analysis has shown that all core modules have the same high level structure as in Fig. 5, and that all their makefiles are customized in the same way in order to link to stub implementations of dependent modules. Thus, they result in a good common look-and-feel in the high level structure of unit tests. Furthermore, developers can also replace stubs by real modules and can perform incremental module integration and validation.

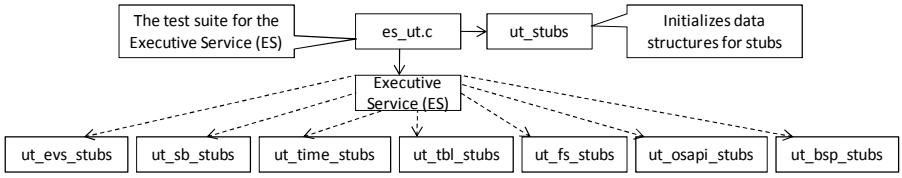


Fig. 5. The High level structure of Unit Tests (example view for the Executive Services (ES) module). Arrows denote dependencies (e.g. calls). Dotted arrows are dependencies established at link time. The ES module is linked to stubs that implement the interfaces of modules ES depends on. The main function is defined in es_ut.c (ES unit test) which runs all test programs implemented in es_ut.c.

Table 2. The number of stub functions used for testing each core module

	Stub SB	Stub ES	Stub EVS	Stub Time	Stub TBL	Stub FS
SB	NA	11	3	1	0	1
ES	10	NA	4	3	1	4
EVS	8	10	NA	1	0	1
Time	9	8	2	NA	0	
TBL	9	15	3	1	NA	3
FS	0	2	0	1	0	NA

How the stubs are designed and implemented. The CFS implements stubs for each of the publicly visible APIs of its modules. The test suite for a specific module uses stub implementations of functions of other modules in order to fully run each function of the module under test (see Table 2) and in order to provide an environment that produces guaranteed results for each possible function call. In order to achieve 100% path coverage of each function under test, developers or testers also need a way to manipulate return values of the stubbed functions. Otherwise, unit tests will take a lot of time to run and it may also be difficult to pinpoint where a test actually failed. Keeping these requirements in mind, the CFS team has defined the data structure in Fig. 6 for unit testing purpose.

```
typedef struct
{
     uint32 count;
     uint32 value;
} UT_SetRtn_t;
```

```
void UT_SetRtnCode (UT_SetRtn_t *varPtr,int32 rtnVal,int32 cnt){
    varPtr->value = rtnVal;
    varPtr->count = cnt;
}
```

Fig. 6. The Key data structure used for controlling return values of functions. Each stub implementation of core module functions has its own instance of this structure. Testers manipulate the instance of this data structure. Stubs are programmed to return values of interest based on the state of the *count* variable. The logic of each stub is based on the state of the *count*. For example, a stub function can be implemented to return 0, if count is positive, and otherwise -1. Fig. 8 shows an example stub function. Right: Setting up the return values for each instance of the UT_SetRnt_t (in ut_stubs.c). The stub implementation for each function returns values based on the state of the count initialized using this function.

The ut_stubs module, shown earlier, creates several instances of the above data structure – one for each stub implementation of the core module functions. It is the responsibility of the tester to write stubs and manipulate return values using the state of *count* variable shown in Fig. 6. Note that all stubs have exactly the same function signature as the real the implementation. This is an important requirement otherwise the source code of the function under test has to be changed in order to unit test it, which is, of course, not a good engineering practice.

Consider the interface specification of the create pipe function of the software bus module, see Fig. 7. The original implementation returns one of four possible return values. However, the original implementation also creates real queues using the OS abstraction layer. If we want to unit test a function defined in another module that uses this create pipe function, the developer or tester should be given an easy way to manipulate return values so that different paths can be traversed easily. Also, in this scenario, the mock implementation does not need to create queues for unit testing of other modules. Such a mock implementation of create pipe is shown in Fig. 8. As we can see, it does not do too much in contrast to the original implementation. Nevertheless, it is remarkably useful from the testing point of view because of the capability it offers to control return values using the SB_CreatePipeRtn instance of the UT_SetRtn_t data structure.

```
/*****************************************************************************
** Name:    CFE_SB_CreatePipe
**
** Purpose: API to create a pipe for receiving messages
** Inputs:
**  PipeIdPtr - Ptr to users empty PipeId variable, to be filled by  this function.
**  Depth     - The depth of the pipe (max number of messages the pipe can hold at any time).
**  PipeName  - The name of the pipe displayed in event messages
**
** Outputs:
**   PipeId    - The handle of the pipe to be used when receiving messages.
**
** Return Values:
**    Status - CFE_SUCCESS, CFE_SB_BAD_ARGUMENT, CFE_SB_MAX_PIPES_MET, CFE_SB_PIPE_CR_ERR
**
******************************************************************************/
int32  CFE_SB_CreatePipe(CFE_SB_PipeId_t *PipeIdPtr, uint16  Depth, char *PipeName)
```

Fig. 7. Interface specification of the create pipe function of the software bus module

```
extern UT_SetRtn_t  SB_CreatePipeRtn;

int32 CFE_SB_CreatePipe (CFE_SB_PipeId_t *PipeIdPtr, uint16 Depth, char *PipeName){
    if (SB_CreatePipeRtn.count > 0)
    {
        SB_CreatePipeRtn.count--;

        if(SB_CreatePipeRtn.count == 0)
        {
            return SB_CreatePipeRtn.value;
        }
    }
    return CFE_SUCCESS;
}
```

Fig. 8. The mock implementation of the create pipe function (in ut_sb_stubs.c file). This example shows how the UT_SetRtn_t data structure is manipulated to return different values.

Suppose we want to force the create pipe to return CFE_SUCCESS, all we need to do is just call the UT_SetRtnCode function as shown in Fig. 9 in our test function. This enables the test program to systematically control return values of other functions in order to traverse different paths of the program under test.

```
// forces CreatePipe to return CFE_SUCCESS
UT_SetRtnCode(&SB_CreatePipeRtn, -1, 2);
```

Fig. 9. Example of forcing a function to return the value of interest. See Fig. 6 (right) for the definition of this UT_SetRtnCode function.

The review has shown that all mocked functions and test programs follow this technique to manipulate return values. In addition, all mock implementations are very small, as little as 10 lines or so. Thus, it indicates that this technique works in practice and requires neither significant learning time nor major shift in the way of working. Table 3 shows that there are dedicated test programs for each public function of each core module. The two ES functions and one TBL function have no unit tests because they are single line get functions. The right hand side of the below table shows that not all internal functions are directly tested. However, further analysis has shown that they are transitively tested using the test programs of the public interfaces. This shows that the stub-based unit test architecture is possible to develop and works well in practice even though stubs and unit tests are manually developed at this point.

Table 3. Left: Interface coverage by unit tests. Right: The total number of functions unit tested directly. Some internals functions are also directly unit tested because they are defined as non-static C functions, otherwise internal functions are transitively tested using public APIs.

Core Module	# of Functions in Interface	# Directly invoked in Unit Tests
SB	30	30
ES	33	31
EVS	7	7
Time	24	24
TBL	14	13
FS	5	5

Core Module	# of Functions Defined	# Directly invoked in Unit Tests
SB	86	45
ES	117	68
EVS	33	12
Time	72	42
TBL	60	41
FS	11	11

Some design issues that make unit testing harder

Consider the code snippet defined in the software bus module (see the right of Fig. 10). It shows that the function returns the same value “bad argument” from two different conditional blocks. As a consequence, the unit testing code of this function becomes slightly more complex than necessary because it needs to determine exactly which one of the two code snippets returned that value. It does so by calling the stub implementation of the send event (similar to logging) function to make sure the number of times it was called is equal to 1 if MsgPtr is null, otherwise 2 if the msg id is invalid. This review identified a few functions that suffer from this design problem with respect to return values. These issues are being addressed by the CFS team. The recommended fix is to change such functions so that they all return a unique return value from each of its path, and thus make the unit testing code clearer.

```
int32  CFE_SB_SendMsg(CFE_SB_Msg_t    *MsgPtr) {
 /* check input parameter */
if(MsgPtr == NULL){
 CFE_EVS_SendEventWithAppID("Send Err:Bad input argument",…);
 return CFE_SB_BAD_ARGUMENT;
}

MsgId = CFE_SB_GetMsgId(MsgPtr);
/* validate the msgid in the message */
if(CFE_SB_ValidateMsgId(MsgId) != CFE_SUCCESS) {
 CFE_EVS_SendEventWithAppID("Send Err:Invalid MsgId", …);
 return CFE_SB_BAD_ARGUMENT;
}
    …
}
```

```
void Test_SendMsg_NullPtr(void){
   …
 ActRtn = CFE_SB_SendMsg(NULL);
 ExpRtn = CFE_SB_BAD_ARGUMENT;
 if(ActRtn != ExpRtn){
    TestStat = CFE_FAIL;
 }

 ExpRtn = 1;
 ActRtn = UT_GetNumEventsSent();
 if(ActRtn != ExpRtn){
    TestStat = CFE_FAIL;
 }
 …
}
```

Fig. 10. An example code snippet that makes unit testing difficult. It shows that the same return code is used for different issues. To unit test this function, the mock implementation of send event function counts the number of times the send event function is called, in order to make sure the correct path is tested for the test data. The right figure shows that the test case has to also test the side effect, that is, the number of logging events it was sent out by the function under test. UT_GetNumEventsSent gets the number of logging event using the data structure manipulated by the mock implementation of the SendEventwithAppId function, which simply counts the number of times it is being called.

Some design decisions that make unit testing easier

Our review of unit tests has derived some insights on the influence of product line architectural design decisions on unit testing. Here, some product line specific examples from the CFS are disclosed.

The key to flexible unit testing is programming to abstract interfaces and moving out conceptually orthogonal variation points to the right module. For example, in the CFS case, the core layer is designed and implemented in such a way that it is completely agnostic to the OS, hardware, and board support packages. More concretely, consider the simple case of creating a queue, and sending and receiving messages using the queue. Naturally, different OSes offer different Queue APIs. If the system is programmed with a hard binding to the OS specific APIs then it is of course very difficult to unit test such a system, and a different sets of unit tests have to be developed for each OS type. In the CFS case, abstract interfaces with diversified implementations are developed, and thus conceptually orthogonal variation points are moved out of the module (see Fig. 11).

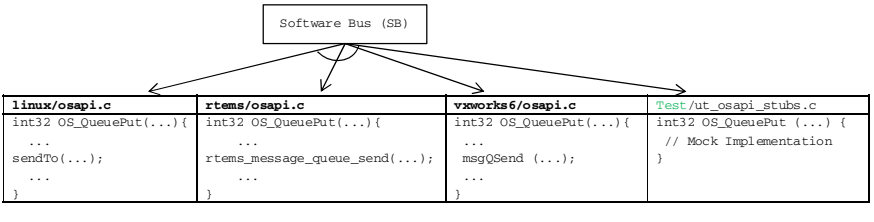


Fig. 11. A common abstract API with different implementations, including a mock implementation for unit testing. The SB module is programmed to the abstract interface, and the actual binding to a specific implementation is only at the link time.

Some internal details of modules should be made public. While hiding modules' secrets is one of the fundamental principles of software engineering [3], this principle has to be weakened in order to write good unit tests. For example, consider the load library function snippet (see Fig. 12), which loads the given shared library (LibName) and calls the function with the given name (EntryPoint). This is defined in the Executive Service module. The CFE_ES_MAX_LIBRARIES is a variation point defined in a public header file that must be set to a particular value. This function should return an error code if it is called more than the number of times set during configuration. Note that it uses the CFE_ES_Global data structure for keeping track of number of libraries that are already loaded. This data structure is hidden inside the ES module, meaning that no other module is allowed to access this data structure or know about it or its details. However, in order to test that this function will return an error code if it is called more than the configured number of times, the unit test must have access to CFE_ES_Global data structure; otherwise it is very difficult to simulate this error scenario. To this end, the CFS designers had made this global variable public to other internal files of the ES module, and thus the unit test can access and manipulate this variable. Architectural rules were defined to make sure such publicly visible secret variables are not referenced by other modules using the approach presented in [2]. This is an example of how the risk of violating some engineering principles can be mitigated by adding architecture/design rules.

```
int32 CFE_ES_LoadLibrary(char *EntryPoint,  char *LibName, …) {
   boolean LibSlotFound = FALSE;
   for ( i = 0; i < CFE_ES_MAX_LIBRARIES; i++ ) {
      if ( CFE_ES_Global.LibTable[i].RecordUsed == FALSE ) {
          LibSlotFound = TRUE;
          break;
      }
   }
   if(LibSlotFound == FALSE) return CFE_ES_ERR_LOAD_LIB;
}
```

Fig. 12. The Load library function loads the library (LibName) and calls a function (EntryPoint) of that library. CFE_ES_MAX_LIBRARIES is a variation point defined in a header file. CFE_ES_Global is a global variable, allowing the unit test to change the state to validate the scenario that if this function is called more than the configured number of times, an error will be returned code (see Fig. 13).

```
/* Test for loading more than max number of libraries */
 for (j= 0; j < CFE_ES_MAX_LIBRARIES; j++) {
        CFE_ES_Global.LibTable[j].RecordUsed = TRUE;
 }
 Return = CFE_ES_LoadLibrary("EntryPoint","LibName", …);
 UT_Report(Return == CFE_ES_ERR_LOAD_LIB, "CFE_ES_LoadLibrary",
                        "No free   library slots");
```

Fig. 13. Test code from the load library function that tests the behavior of the load library function when it is called more than the allowed number of times (CFE_ES_MAX_LIBRARIES) (see Fig. 12). It manipulates the ES module's internal data structure that keeps track of the number of loaded libraries.

Table 4. Answers to questions based on the analysis

Question	Answer and Comments
1. Can a core module be tested independently of core modules it uses?	Yes. Because of the novel design of simple stubs, it only takes 3 minutes or so to run all the unit tests of the core modules.
2. How are variation points of each module being handled during unit testing?	There is a unit test program for each variation point that checks the behavior for upper and lower bound constraints. Some internal details of a module are made public to support unit testing.
3. How easy is it to create mock or stub implementations of dependent modules?	Mock or stub implementations are easy to create. At link-time, a module can be linked to one or more stubs of dependent modules. This capability supports incremental integration too.
4. Can modules be unit tested without access to special hw and OS?	Yes. Testers can test on their desktop and do not need to go to the test lab for unit testing. The UTF framework provides simulators with the same API as the original code.
5. How easy it is to set-up unit tests for a module?	Just a couple of instructions are needed to set-up a test program.
6. Are there dedicated tests for each public function of a module?	Yes, all interfaces have one or more dedicated unit test programs.
7. How lengthy and complex is each test program?	Some are lengthy (~100 lines) because they test more than one scenario and could be split into smaller ones. Some are complex because the function under test returns the same return code from multiple paths requiring extra test code.
8. How are the tests results collected and reported for further analysis?	Currently, they use the gcov (GNU coverage) line coverage tool. All failures are reported in a text file that is manually reviewed by the tester.
9. Is there a common look-and-feel in the way the modules are unit tested?	Yes. All core modules consistently use the concept of stubs to do unit testing. Also, all test makefiles for test suites share the same structure.

5 Closing Remarks

In this paper, we described the analysis of the CFS product lines' unit testing strategy and accompanying unit test cases and testing environment. The CFS has been refined over more than 10 years and has gone through rigorous inspections and improvement initiatives. In addition, the CFS captures knowledge from implementing dependable flight software for more than 20 years of specifying, developing, testing and flying such software. Thus, we are grateful that we can analyze and use CFS as an example of good software engineering that we can all learn from, even though there are still

some issues that can be removed and improved. For example, the CFS has tackled the difficult practical unit testing problem that modules often depend on other modules, making them hard to separate and unit test in an independent fashion. In addition, modules can also depend on unique features and functions provided by the operating systems, and they may require the hardware in-the-loop for the software to function properly, making it difficult to set up a controlled unit test environment. The CFS team's approach to unit testing also handles the use of modules (real modules or stubs for testing) as a set of variation points. This introduces a level of flexibility that allows the user of CFS to also use the same set up for incremental integration testing because stubs for testing can be swapped in or out depending on the situation, thus limiting the risks that are associated with big bang integration testing. A future paper will disclose the unit testing framework that allows application developers to unit test their applications without running the core modules. Unit testing and the type of incremental integration testing described above are only two aspects of testing, and other forms of testing needs to be conducted in order to detect those types of defects that such testing cannot detect. Supported by the NASA IV&V center, Fraunhofer, in collaboration with the CFS team and using CFS as a testbed, are researching ways to develop new testing techniques that address these challenges.

Acknowledgments. Lisa Montgomery and her NASA IV&V team and Sally Godfrey NASA GSFC for supporting this work; Charles Wildermann, all members of the GSFC CFS team for comments and discussions; Rene Krikhaar for the RPA toolkit; The Prefuse visualization team, at Stanford University, for making it available to us; Lyly Yonkwa for fruitful discussions; three anonymous reviewers for comments.

References

1. Feijs, L., Krikhaar, R., Van Ommering, R.: A Relational Approach to Support Software Architecture Analysis. Software Practice and Experience 28(4), 371–400 (1998)
2. Ganesan, D., Lindvall, Ackermann, C.M., McComas, D., Bartholomew, M.: Verifying Architectural Design Rules of the Flight Software Product Line. In: SPLC (2009)
3. Hoffman, D., Weiss, D.: Software Fundamentals – Collected Papers of David L. Parnas. Addison-Wesley Publications, Reading (2001)
4. The OS Abstraction Layer of the CFS, `http://opensource.gsfc.nasa.gov`

Sans Constraints? Feature Diagrams vs. Feature Models

Yossi Gil*, Shiri Kremer-Davidson, and Itay Maman

IBM Research—Haifa

Abstract. In this paper we study *constraints*—inter-dependencies between basic features in a feature model which are not captured by diagrams. We offer a method for the removal of these constraints and explain why their removal require an (inevitable) exponential increase to the tree size. We show that the elimination of constraints makes it possible to provide an efficient solution for the feature editing problem, recently raised by Thüm, Batory and Kästner. We tie feature models with computer science fields which may appear very foreign to our domain, including circuit complexity, graph algorithms and algebraic complexity. The objective of this tie is double folded: drawing the attention of the foreign community to the problems we address in our field, and to suggest the use of current results in these fields for better understanding of the mathematics behind the modeling of software product lines.

1 Introduction

Ever since "Feature-Oriented Domain Analysis (FODA)—Feasibility Study", the seminal work of Kang et al. [9], people tend to think of variability in software product lines (SPL) in terms of *feature models*. A compact representation of commonalities and variability of all members of a given software product line is achieved by erecting a feature model. This model defines a set of *basic features*, while each valid combination of these features defines a member of the product family. Feature models are used extensively during the development process, and often control the production and assembly of pieces of code and other artifacts to realize any of the members of a family.

Following FODA, features are organized in a tree structure, in which the parental relationship *represents* a logical grouping of features. We shall call this graphical organization of the interaction between features [22] in a hierarchical fashion, a *feature diagram*, where leaves will be referred to as *basic features*. The internal nodes are of several kinds, typically *mandatory-*, *optional-*, *or-* and *alternative-* nodes. FODA introduces also visualization aids: Connections between a feature and its group of children are distinguished as And (no arc), Or (filled arc) and Xor (unfilled arc). The children of And-groups can be either mandatory (filled circle) or optional (unfilled circle). The particular notations

* On sabbatical from the Technion—Israel's institute of Technology

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 271–285, 2010.

will be of a lesser importance here. The more crucial property is that no basic feature occurs in the tree more than once.

A *feature model* is more general than a feature diagram, in that it augments this tree with additional *cross tree constraints*, which express interaction which went uncaptured by the diagram. The cross tree constraints are written as logical formulas of arbitrary complexity over the basic features.

Cross tree constraints give feature models maximal expressive power, since any interdependency between the basic features may be thus captured. Moreover, even a simple logical condition such as

$$(\neg a \wedge b) \vee (b \wedge c) \vee (c \wedge d) \tag{1}$$

cannot be written as a constraints-free feature diagram. (See Section 3.2 below.)

On the other hand, the tree form is particularly amenable to logical maneuvering that in the general case require complex *Logic Truth Maintenance Systems* (LTMS) [2]. The reason is that in the tree structure, decisions made in two disjoint subtrees are orthogonal: they only interact at a single point, the least common ancestor of the roots of these subtrees.

1.1 Cost and Utility of Constraints

The main question intriguing us in writing this paper is that of a better understanding of the issue of *orhogonalization*: what are the circumstances in which *cross tree constraints* can be eliminated and what is the cost of doing so. In other words, we strive for a better understanding of the difference between feature models and feature diagrams. En route, we shall try to offer a better understanding of the the kinds of operators used in a feature diagram, that is, the kinds of their internal nodes, mandatory, optional, etc.

Our study links feature diagrams to notions in the theory of computational complexity. As it is often the case in this field, most results are not encouraging. A lower bound proof cannot be integrated as a novel component in our next shining CASE tool. Still, our theoretical work does offer a better comprehension of the things we can and cannot do.

In a sense, our work is complementary to that of Czarnecki and Wąsowski [4] who gave a heuristic for translating any logical formula into a feature model, except that this heuristic did not guarantee that the generated model will not contain cross tree constraints. We show that that one cannot hope for anything better in the settings that Czarnecki and Wąsowski considered, i.e., a direct translation of a general formula into a tree structure. We will relax this condition, and will consider translations in which the variables of the original formula are "mapped" into a new set of basic features, and discuss the inherent cost of doing that.

We shall argue that constraints are inevitable, since the expressive power, that is, the number of different statements which can be made, of diagrams is much more limited than that of models. Moreover, we explain how the introduction of of *compound features* necessitates the introduction of constraints.

On the other hand, we show that at the cost of blowup of either the diagram size, or the translation map, all constraints can be eliminated. One of our surprising findings is that recent results in algebraic complexity can be used for more efficient solution of the problem of edits to a sans constraints feature model [18]; this solution applies also in the case that all constraints are of the sort required for representing compound features.

1.2 Feature Diagram Kinds

Our study will consider a number of different kinds of feature diagram. Of the least expressive power is the *Monotone kind*, in which tree nodes are restricted to be AND and OR only. This somewhat restrictive model may be convenient to use in the case that feature interaction is minimal.

Next is the familiar *Logical kind*, in which the nodes may be any of these three operators: OR, AND, or XOR. Following that, we have the *Polynomial kind*, in which nodes are restricted to AND and XOR, which will turn out to be surprisingly interesting. On the one hand, we shall see that it is sufficiently expressive to accommodate any logical formula. On the other hand, it will be shown that it has efficient general solutions for some problems that were considered intractable in the SPL literature, and hence requiring various heuristics.

Next in the hierarchy is what we call the *Foda kind*, which augments the *Polynomial kind* with a SELECT operator, which in a sense is a k-ary XOR, in that it yields **1** only in the case that exactly one of the inputs is **1**.

However, several enhancements to this basic notations were proposed in the literature, including n-out-of-m nodes, at least k nodes [3], etc. To reason about these variants we introduce the *Unrestricted kind* of feature diagrams, in which an arbitrary, yet bounded degree, operators are allowed in any tree node. A particular subkind of the Unrestricted kind may be defined to cover interesting cases which cannot be represented by the Foda kind. For example, one may introduce an operator that captures the recalcitrant example of (1). Unfortunately, the introduction of new operators will never be sufficient. As hinted above, we show that there are exponentially many conditions that will not be expressible even in Unrestricted feature diagrams, regardless of the set of operators it offers.

Table 1 is a chart of the domain of our interest, breaking it down by diagram kinds. Columns denote the operations that one may be interested in with respect to each of these model. The first four content columns are manipulations intrinsic to the SPL realm. The last three columns denote operations for the elimination of constraints, i.e., for the conversion of general logical formulae into the special form dictated by the model.

Each cell content denotes the time complexity of the perspective operation, using the familiar $\mathcal{O}$ notation, i.e., $\mathcal{O}(n)$ is linear, $\mathcal{O}(n^2)$ is quadratic, $n^{\mathcal{O}(1)}$ is polynomial, and $2^{\mathcal{O}(n)}$ is exponential. A question mark denotes that even though the complexity is unknown, we have a good guess of the complexity;

The large number of question marks says that the domain is still largely terra incognita. The importance of drawing this map is greater than charting the

Table 1. Constraints in different kinds of feature diagrams

Kind	*Satisfiability*	*Closure*	*Equivalence*	*Inclusion*	*Detection*	*Refactoring*	*Mapping*
Monotone	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	?	$\mathcal{O}(n^2)$	?
Logical	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	?	NP-HARD	?
Polynomial	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	?	NP-HARD[a]	$2^{\mathcal{O}(n)}$
Foda	$\mathcal{O}(n)$	$\mathcal{O}(n)$	?	?	?	?	$2^{\mathcal{O}(n)}$
Unrestricted	$\mathcal{O}(n)$	$\mathcal{O}(n)$	?	?	?	?	$2^{\mathcal{O}(n)}$

[a] If the input is provided in a multilinear format then only $\mathcal{O}(n \lg^2 n)$.

unknowns (with the hope of drawing attention of the computational complexity community to our needs).

1.3 SPL Problems

Continuing with our description of Table 1, ***S****atisfiability* is the problem of determining whether a model has a product in it. If arbitrary constraints are allowed, then the problem is intractable, since it is tantamount to the NP-complete 3SAT problem. In contrast, with all kinds of feature diagrams, not only the basic problem is linear time, but also is its more general variant, in which it is asked whether a product exists even in face of partial decisions on some of the features.

The more general ***C****losure* problem is that of determining all consequences of a partial decision on some of the features. The difficulty lies with that decisions can be made not only on basic features, but also on internal nodes. In both the Monotone and the Polynomial kinds, we have a representation of the consequences of partial decisions, which is not only compact, but also amenable to further manipulations.

In the more general kinds, a compact representation is easy to achieve; all you do is conjunct the assignments to the partial decisions with the tree structure. However, it is not clear that this representation is useful for further manipulations.

The ***E****quivalence* problem is the one raised by Thüm, Batory and Kästner in their quest [18] for a method to determine whether an edit of a feature model changed the set of permissible products. In face of constraints, the problem is obviously intractable, and there is little wonder why these authors had to resort to heuristics in tackling it. However, as indicated by the table, there is a linear time algorithm for doing that in the Monotone, Logical, and Polynomial kinds. We suspect that a same complexity algorithm exists even for the Foda kind. We do not dare making such a guess for the Unrestricted kind, but as Thüm et al. observed, the problem can always be reduced to a satisfiability problem. (A converse reduction is not known so, we cannot say the problem is NP-complete.)

Next is the related ***I****nclusion* problem, also raised by Thüm et al., which is to determine whether the set of permissible products of one feature model is contained in that of another. Again, a non-heuristic solution is hopeless in the case that constraints are involved. Perhaps surprisingly, in the Polynomial kind (and hence in all inferior kinds), the problem cannot be reduced to that of divisibility of multilinear multivariate polynomials, which can be carried out in

linear time (even in the face of compound-feature type of constraints). We offer no guess as of the complexity of the problem in more general kinds.

1.4 Representability in Feature Diagrams?

The final bulk of three columns in Table 1 is concerned with issues involved in the elimination of constraints. Given a set of conditions on the basic feature set, we ask the following questions: ***D****etection*: can one determine whether they can be arranged as a feature diagram? ***R****efactoring*: If this is indeed possible, how can this process be carried out? Finally, if such a refactoring is impossible, we ask the ***M****apping* question: is there a way of "mapping" the basic features into a different set of basic features which can be arranged in a diagrammatically form, while preserving the "essence" of the original problem.

As it turns out, there are elegant combinatorial characterization of the cases which can be described as feature diagrams of the Monotone and Logical kinds. In the case of Monotone, a fairly efficient refactoring algorithm was recently discovered [6]. Unfortunately, no such algorithm is known in the Logical kind, and all we have is a brute force method of trying out all possible feature diagrams against a given input, with the hope that one will do. This is essentially the case also for the other kinds.

The Polynomial kind is interesting in this respect. There is an elegant algorithm, using polynomial factorization algorithms for detecting whether a set of constraints can be written in a Polynomial diagram. Moreover, if the input is given in a special, "reduced" form, which occurs sometimes, then refactoring can be done in $\mathcal{O}(n \lg^2 n)$ time.

The final column in the table deals with the issue of translating a general set of constraints into a feature model, by a process which can be thought of as variable substitution in algebra. We show a method for devising such a substitution in the higher kinds. Admittedly, the substitution is exponential in the sense that it leads an exponential blowup in the size of the diagram, or in the size of substitution. But, as as we argue below, nothing better is possible, since feature diagrams are exponentially less expressive than feature models. Also, a more efficient method for translation would lead to sub-exponential algorithm for NP hard problems, an achievement that we cannot hope for...

Perhaps a conciliating thought is that if the constraints are small with respect to the original model, i.e., of logarithmic size, then an exponential price can still be tolerated.

Outline. *The remainder of this article is organized as follows. Section 2 gives some definitions and explains how compound features are to be represented as constraints. We elaborate the topic of limitations of the expressive power of feature diagrams in Section 3. Section 4 gives special attention to the Polynomial kind of feature diagrams. Section 5 concludes and lays out directions for future research. Due to space limitations, we do not give equal attention to all cells of Table 1, and proofs are sketchy at best. Also, the paper is largely self contained and should be accessible to general, not necessarily theoretical computer science community.*

2 Definitions

The representation of feature models as propositional calculus is well known [23,2,16]. Here, we will redo this briefly, dwelling in the (equivalent) realm of *Boolean algebra*. Recall that *Boolean algebra* has *Boolean expressions* (henceforth, *expressions* for short) defined by an alpha set $\mathbf{A}$ of *variables* which form the *atomic* expressions, and the closure of the atomic expressions with *binary* operators: conjunction (which we will sometimes write as multiplication), disjunction, implication, exclusive-or (which we will write as $\oplus$), *unary* negation and the *nullary* operators (constants) $\mathbf{0}$ and $\mathbf{1}$.

The Foda kind also uses a k-ary SELECT operator which we will write as ς; the expression $\varsigma(E_1, \ldots, E_k)$ is thus $\mathbf{1}$ when one, and only one, of $E_1, \ldots, E_k$ is $\mathbf{1}$. Observe that $\varsigma(E_1, E_2) = E_1 \oplus E_2$, but $\varsigma(E_1, \ldots, E_k) \neq E_1 \oplus \cdots \oplus E_k$ for $k > 2$. There is no need to have a special operator for *optional* features. These can be written as a simple XOR with a dummy, "not existing" basic feature. Other operators may be introduced as well.

The *size* of an expression E is simply the number of nodes in its tree representation, that is 1 if E is a variable, or $1+|E_1|+\cdots+|E_k|$ if $E = \psi(E_1, \ldots, E_k)$ and ψ is a k-ary operator.

2.1 Feature Diagrams over Basic Features

Now, a feature diagram is an expression in which no variable (basic feature) participates more than once. More formally,

Definition 1 (Feature Diagrams). *Let Ψ be a finite set of Boolean operators. A* feature diagram over Ψ *is either a variable $a \in A$, or an expression of the form $\psi(F_1, \ldots, F_k)$ where $\psi \in \Psi$ is a k-ary operator, and $F_1, \ldots, F_k$ are feature diagrams over Ψ, such that the sets of variables used in $F_1, \ldots, F_k$ are disjoint.*

(Observe that this recursive definition does not offer a special handling of the constants $\{\mathbf{0}, \mathbf{1}\}$. The perspective by which these are nullary operators is sufficient. With this perspective, the set of operators $\{\text{XOR}, \text{NOT}\}$ is "equivalent" to the operators set $\{\text{XOR}, \mathbf{1}\}$.)

Now, the Monotone feature kind is simply feature diagrams over the operators set $\Psi = \{\text{AND}, \text{OR}\}$; in the Logical kind, $\Psi = \{\text{AND}, \text{OR}, \text{NOT}\}$; the Polynomial kind is defined by $\Psi = \{\text{AND}, \text{XOR}, \mathbf{1}\}$. In the Foda kind, the set of operators is infinite (yet "regular") $\Psi = \{\text{AND}, \varsigma_2, \varsigma_3, \ldots\}$. The Unrestricted kind of feature diagrams, does not say anything about the set Ψ, other than the demand that it is finite, and hence of bounded arity.

An *assignment* (respectively, a *partial assignment*) α to a set of variables $V \subseteq \mathbf{A}$, is a function (respectively, a partial function) $\alpha : V \to \{\mathbf{0}, \mathbf{1}\}$. If V is the set of all variables participating in an expression E, then an assignment to V *satisfies* E if E evaluates to $\mathbf{1}$ under this assignment; a partial assignment is *consistent* with E if it can be *completed* into a satisfying assignment. The *closure* $\alpha(E)$ of a partial assignment α consistent with expression E, is a superset of α consisting of the assignments to variables in E which are common to all satisfying completions of α.

Any specific feature diagram identifies the set of satisfying assignments to its variables. In the software product lines world, a feature diagram represents commonalities and variability of a product line, while its set of satisfying assignments is isomorphic to the set of products.

2.2 Feature Diagrams with Compound Features

The SPL literature often speaks of *compound* features. For example, Fig. 1 depicts a feature diagram of a hypothetical product line of cellular phones. Intuitively, the figure says:

"When designing a particular cellular *phone, make a* color *decision and a* hardware *decision. Color is either* red *or* blue *(but not both). Hardware must include a* dialing *unit,* and and *an accessory which is either a* bluetooth *or a* camera, *which if included, is either a one- or a three- mega pixel.*

It is tempting to apply a straightforward translation of this tree into an expression, converting every XOR node into a $\oplus$ and every AND node into conjunction, obtaining

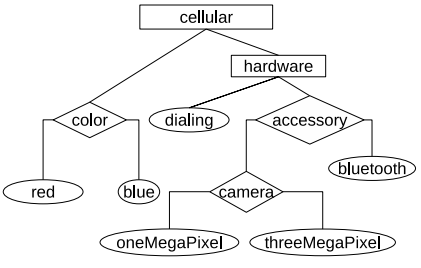


Fig. 1. A feature diagram of a cellular phone product line

$$(\mathsf{red} \oplus \mathsf{blue}) \wedge \Big(\mathsf{dialing} \wedge \big((\mathsf{oneMegaPixel} \oplus \mathsf{threeMegaPixel}) \oplus \mathsf{bluetooth}\big)\Big).$$

Obviously, this expression is *not* equivalent to the diagram. For instance, it allows decisions where oneMegaPixel, threeMegaPixel and bluetooth are simultaneously included. The difficulty is that such a translation does not introduce variables for the internal nodes, thereby making it cumbersome to express the restrictions induced by the structure of the original tree (diagram).

The remedy is simple. First, augment any k-ary operator ψ with its $k+1$-ary *labeled variant* ψ', defined by $\psi'(\ell, E_1, \ldots, E_k) = \ell \wedge \psi(E_1, \ldots, E_k)$, and second, introduce a constraint

$$\neg\ell \rightarrow (\neg E_1 \wedge \cdots \wedge \neg E_k). \tag{2}$$

Thus, ψ' yields **1** only when the labeling variable ℓ, which is nothing but a compound feature, is **1**. However, when the compound feature is **0** all arguments of ψ must be **0**.

A nice property of the constraint (2) is that it is local, i.e., *not* cross-tree. The fact that it only adds another dependency between a node and its children makes it possible to devise efficient algorithms also when compound features are present.

Also note that the constraint (2) makes sense only for "**0**-funneling" operators, i.e., operators which are **0** when all their parameters are **0**. We will not be interested in non **0**-funneling operators.

3 Expressive Power of Feature Diagrams

The reason that pure feature diagrams are so appealing is that satisfiability and closure are easily solvable in these.

Theorem 1 (Satisfiability and Closure). *There exists a linear time and space algorithm which, given an* Unrestricted *labeled feature diagram F and a partial assignment α to its basic and compound features, determines*

1. *Whether α is a consistent assignment to F.*
2. *The closure assignment $\alpha(F)$, that is the set of all assignments which are forced by α.*

Proof. (Sketch.) The algorithm first traverses F bottom up, assigning a $\{\mathbf{0}, \mathbf{1}, ?\}$ value to each node in it. In reaching an internal node, it checks whether the values coming from its children are such that the condition implied by the operator in this node can be satisfied according to these values. It further examines whether α has prescribed a value for this internal node, and whether this value is consistent with the children's' values.

In the second phase, the algorithm traverses the tree top-down, propagating with it from each node it visits any implications it has on the values of its children. □

We can also show that there are no other implications of α other than the $\{\mathbf{0}, \mathbf{1}, ?\}$ assignments that the algorithm makes. In a sense, this algorithm generates another simpler feature diagram $F' = F'(\alpha)$. Also, the algorithm can be made to suggest corrections to α in the case that it is inconsistent with F, and can be strengthened to deal with arbitrary local constraints, rather than the limited form ([2]), provided of course that each node has a bounded number of children.

3.1 Unrestricted Feature Diagrams

On the negative side, we note that the expressive power of feature diagrams is inherently restricted. To see this, consider the total number of Boolean functions $\{\mathbf{0}, \mathbf{1}\}^n \to \{\mathbf{0}, \mathbf{1}\}$. Since the number of distinct inputs such a function has is 2^n, and each of this is either $\mathbf{0}$ or $\mathbf{1}$, there are 2^{2^n} such functions. On the other hand, the number of distinct trees over n variables is in $2^{\mathcal{O}(n \log n)}$. (Any such tree can be written in $\mathcal{O}(n)$ memory cells of $\mathcal{O}(\lg n)$ bits each—cells which will contain the variable symbols, operator symbols and pointers used for representing this tree.)

It follows that only an exponentially small fraction of all Boolean functions can be represented as feature diagrams over the same set of variables. And, it should be easy to be convinced that this predicament is not made any better by allowing constraints over compound features, since their total number is linear in the number of basic features.

3.2 Logical Feature Diagrams

Recall that Logical feature diagrams are those that use AND, OR and NOT operators. As it turns out, the expressive power of Logical feature diagrams is

even more restricted than the above counting argument suggests. First, there are very simple Boolean function which cannot be computed by a tree requiring that each variable participates only once.

Theorem 2 ([15]). *Let f be the Boolean function of n variables whose value is* **1***, whenever 2 or more of the inputs are* **1***. Then, the size of any expression which computes f using* AND*,* OR *and* NOT *operators is in $\Omega(n \lg n)$.*

Recall that the set $\{\text{AND}, \text{OR}, \text{NOT}\}$ is a basis for Boolean algebra, i.e., any other bounded arity operator can be represented with a constant number of applications of these. One may try to conclude from this that there is no feature diagram which computes function f, regardless of the content of Ψ. While being probably true, this conjecture is an open question. The difficulty lies with the fact in using these three operators to substitute other, more general operators, some of the inputs may need to be replicated, thus invalidating the techniques used in the proof of Theorem 1.

Karchmer, Linial, Newman, Saks and Wigderson [10] gave a complete combinatorial characterization of Boolean functions which can be computed by Logical feature diagrams (called *read once formulae* in the computation complexity jargon).

Let f be a Boolean function over a set of variables V. We say that the pair of sets $\langle U_0, U_1 \rangle$, where $U_0, U_1 \subseteq V$ is a **1**-*witness of* f if setting all variables in U_0 to **0** and all variables in U_1 to **1** forces f to be **1**. The pair is a *minterm of* f if in addition, any other **1**-witness of f, $\langle U_0', U_1' \rangle$, such that $U_0' \subseteq U_0$ and $U_1' \subseteq U_1$, satisfies $U_0' = U_0$, and $U_1' = U_1$.

Consider for example the Boolean function h defined by

$$h(a, b, c, d) = (\neg a \wedge b) \vee (b \wedge c) \vee (c \wedge d). \tag{3}$$

Then, $\langle \{d\}, \{a, b, c\} \rangle$ is a **1**-witness of h, while $\langle \varnothing, \{b, c\} \rangle$ is a minterm of h.

In a similar fashion, we can define **0**-*witnesses of* f, except that it forces the value of f to **0**, and along the same lines, the *maxterms* of f as those **0**-witnesses of f in which the two sets are as small as they can be. For example, for the function h defined in (3), we have that $\langle \{b, c\}, \{d\} \rangle$ is a **0**-witness (since setting $\{b, c\}$ to **0** and d to **1** ensures that the value of h is **0** regardless of the value of the remaining variable a). We also have that $\langle \{b, c\}, \varnothing \rangle$ is a maxterm of h.

Theorem 3 (Detection [10]). *Function f can be represented in a Logical feature diagram if and only if it holds that the intersection of the variables participating in an arbitrary minterm of f with the variables participating in an arbitrary maxterm of f, contains precisely one variable.*

In the example (3), the intersection of the variables in the minterm $\langle \varnothing, \{b, c\} \rangle$ and the variables in the maxterm $\langle \{b, c\}, \varnothing \rangle$ is the two element set $\{b, c\}$, and hence, even the simple function defined in (3) cannot be written as a feature diagram.

Also, we can now show that Polynomial feature diagrams are strictly more expressive than Logical ones.

Theorem 4 (Logical vs. Polynomial). *Every Logical feature diagram F can be rewritten as an equivalent Polynomial feature diagram with F', with $|F'| \in \mathcal{O}(|F|)$. The converse is however false.*

Proof. The OR operator can be written using XOR while referring once to its arguments by the equality $A \vee B = \mathbf{1} \oplus ((\mathbf{1} \oplus A) \wedge (\mathbf{1} \oplus B))$. Conversely, in examining the XOR operator in the spectacles of Theorem 3, we see that $\langle \{A\}, \{B\} \rangle$ is a minterm of $A \wedge \neg B \vee \neg A \wedge B$, while $\langle \{A, B\}, \varnothing \rangle$ is one of its maxterms. However, the intersection of the sets involved in these two terms is of size 2. □

Notwithstanding its elegance, Theorem 3 is not likely to give a procedure for refactoring.

Theorem 5 (Refactoring). *Given a Boolean expression E, $|E| = n$ which meets the conditions of Theorem 3, the problem of rewriting it as a Logical feature diagram is NP-hard.*

Proof. Let S be an instance of 3SAT. Then, S is not satisfiable if and only if $E = \neg S$ is a tautology. However, if E is a tautology, then it meets the conditions of of Theorem 3, and a refactoring procedure, would have resulted in a degenerate tree with a single **1** node, whereby resolving the problem of satisfiability of S. □

Note that Theorem 5 does not imply that there is no effective procedure for ***D****etection*. We conjecture however that no such procedure exists.

Unfortunately, the literature does not offer an equivalent of Theorem 3 for the Foda- and Unrestricted kinds.

We do not know whether there is a detection, or even a nice characterization, of formulae that can be rewritten in the Foda form or in the Unrestricted form for a particular selection of Ψ. However, it follows from the combination of theorems 1 and 5 that the refactoring problem for the Polynomial, Foda and Unrestricted kinds is NP-hard as well.

3.3 Monotone Feature Diagrams

It is well known that monotone Boolean expressions, i.e., expressions which involve AND and OR only, are not as expressive as general expressions (see e.g., [1,11]). We briefly attend to these, demonstrating that despite that, the dealing with constraints even in this setting is not much easier. First, note that the same counting argument with which this section begun, applies. It is mathematical folk-lore [5] that the number of Boolean monotone functions of n variables is $> 2^{\binom{n}{\lfloor \frac{n}{2} \rfloor}} \in 2^{\Omega(2^{n/2})}$, so there is no hope of representing all such functions in Monotone feature diagrams.

There is a characterization of monotone functions which can be written as Monotone feature diagrams, similar to, and even simpler than Theorem 3 (in mintrerms, $U_0 = \varnothing$, in maxterms, $U_1 = \varnothing$). Also, NP-hard issues of the sort of Theorem 5 do not apply.

Still, there is no known effective procedure which given a monotone formula decides whether it can be written as an Monotone feature diagram. All we have is a quadratic time algorithm which takes an expanded formula and converts it into a feature diagram if possible [6].

4 Polynomial Feature Diagrams

The Galois field of size two, denoted by $\mathbb{F}_2$ has the elements $\{\mathbf{0}, \mathbf{1}\}$ and two operations, addition (which is nothing but our XOR function) and multiplication (a logical AND). A feature diagram that is restricted to use these two operations therefore computes a *multivariate polynomial* (polynomials for short) in this field. For completeness, we allow the use of the constant $\mathbf{1}$, which allows for arithmetical negation (just as logical negation).

Polynomials in $\mathbb{F}_2$ can be treated either formally, that is as ordinary polynomials except for the fact that their coefficients are in $\{\mathbf{0}, \mathbf{1}\}$, or as *functions* from $\{\mathbf{0}, \mathbf{1}\}^n$ to $\{\mathbf{0}, \mathbf{1}\}$, where n is the number of variables participating in the polynomial. In the latter interpretation, the polynomial can be simplified by using the equation $x^2 = x$, which holds for all $x \in \mathbb{F}_2$. Thus, if polynomials are treated as functions, interest is restricted to *multilinear* polynomials, i.e., polynomials in which the degree of each variable in each of its monomials is 1.

We are of course interested in the multilinear form, and luckily Polynomial feature diagrams compute precisely that, since each variable can participate at most once in any multiplication. This is not the situation with constraints, i.e., general logical formulae, which by allowing more than one inspection of a variable, may generate formal polynomials. These formal polynomials can of course be brought into a multilinear form, but this operation may be costly, since for doing that, the polynomials need to be expanded.

Theorem 6 (Reed Muller Expansion). *Every Boolean function has a unique representation as multilinear polynomial with no negative terms.*

Proof. See e.g., [13, pp. 58-61]. □

4.1 Refactoring

An elegant property of multilinear polynomials is that they have a unique factorization into factors which are disjoint in their set of variables. This factorization is precisely our ***R****efactoring* operation, what we need in order to express such polynomials as feature diagrams. The ***D****etection* problem is similar. If a multilinear polynomial cannot be factored, then it cannot be represented as a feature diagram.

Theorem 7 (Polynomial vs. Foda). *There exists a feature diagram $F \in$ Foda such that $F \notin$ Polynomial.*

Proof. Applying Reed Muller expansion we have $\varsigma(A, B, C) = A \oplus B \oplus C \oplus ABC$, which cannot be factored into variable disjoint factors. □

McGeer and Brayton [12], described an $\mathcal{O}(n \lg^2 n)$ time algorithm which given a multilinear polynomial, either produces its unique factorization, or announces that no such factorization exists.

Thus, we have a procedure for determining whether a logical formula can be written as a Polynomial feature diagram. To do so, first expand the formula into its Reed-Muller expansion, and then apply the McGeer and Brayton algorithm to produce a feature diagram re-factorization if it exists.

The above process is exponential time, since the expansion may span exponentially many multilinear monomials. Interestingly, there are algorithms that factor $\mathbb{F}_2$ polynomials even without making this expansion [8]. But, this algorithms do so with the formal interpretation. A factorization without expansion in the multilinear interpretation is unlikely, as follows from the proof of Theorem 5.

4.2 Mapping Constraints into Polynomial Feature Diagrams

Table 2 enumerates the 8 assignment to variables a, b, c and d which satisfy the expression $E = (\neg a \wedge b) \vee (b \wedge c) \vee (c \wedge d)$. (This is the same as function h above (3)). The table also introduces 8 *indicator* variables $i_1, \ldots, i_8$ for each of these assignments.

We can define the indicator variables in terms of the original variables,

$$\begin{aligned} i_1 &= \neg a \wedge \neg b \wedge c \wedge d \\ i_2 &= \neg a \wedge b \wedge \neg c \wedge d \\ &\vdots \quad\quad \vdots \\ i_8 &= a \wedge b \wedge c \wedge d \end{aligned} \tag{4}$$

Furthermore, the original variables can be defined in terms of the indicator variables. This definition is done by enumerating, for each original variable, the indicator variables in which it participates in a non-negated form. For example, for variable a these indicator variables are i_6, i_7, i_8. We write therefore,

$$\begin{aligned} a &= i_6 \vee i_7 \vee i_8 \\ b &= i_2 \vee i_3 \vee i_4 \vee i_5 \vee i_7 \vee i_8 \\ c &= i_1 \vee i_4 \vee i_5 \vee i_6 \vee i_7 \vee i_8 \\ d &= i_1 \vee i_3 \vee i_5 \vee i_6 \vee i_8 \end{aligned} \tag{5}$$

This example shows that we have a *general* process of computing a mapping such as (4) that rewrites a logical condition in terms of indicator variables, and an inverse mapping, such as (5), that computes the original variables from the indicator variables. Observe that the original condition (3) can now be written as a simple *labeled* Polynomial feature diagram

$$\ell_1\Big(\ell_3(i_1 \oplus i_2) \oplus \ell_4(i_3 \oplus i_4)\Big) \oplus \ell_2\Big(\ell_5(i_5 \oplus i_6) \oplus \ell_6(i_7 \oplus i_8)\Big) \tag{6}$$

To see why the above ensures that precisely one of $i_1, \ldots, i_8$ is **1**, recall that the use of labels forces all variables in the subtree under the label to **0** if the label

Table 2. Assignments to a, b, c and d which satisfy $(\neg a \wedge b) \vee (b \wedge c) \vee (c \wedge d)$

	a	b	c	d
i_1	**0**	**0**	**1**	**1**
i_2	**0**	**1**	**0**	**0**

	a	b	c	d
i_3	**0**	**1**	**0**	**1**
i_4	**0**	**1**	**1**	**0**

	a	b	c	d
i_5	**0**	**1**	**1**	**1**
i_6	**1**	**0**	**1**	**1**

	a	b	c	d
i_7	**1**	**1**	**1**	**0**
i_8	**1**	**1**	**1**	**1**

is **0**. Thus, if the outermost $\oplus$ evaluates to **1** then *either* ℓ_1 or ℓ_2 is **0**. Say that ℓ_2 is **0**, then so are $i_5, \ldots, i_8$. In this case, ℓ_1 is **1**, and hence one of ℓ_3 and ℓ_4 is **0**. If (say) ℓ_3 is **0** then so are i_1 and i_2, while precisely one of i_3 and i_4 is true. We have thus obtained,

Theorem 8 (Elimination of Constraints I). *Let E be an arbitrary Boolean expression over a variables' set J. Then, there is a labeled* Polynomial *feature diagram over variables set I, $|I| \in 2^{\mathcal{O}(|J|)}$, and mapping expressions $i = E_i(J)$ for all $i \in I$ and $j = E_j(I)$ for all $j \in J$, which define a bijective mapping between the satisfying assignment of E and the satisfying assignments of F. Furthermore, the expressions E_i and E_j are* Logical*, and such that each variable in these occurs only once.*

Proof. As per the above example. □

As mentioned in Section 2, constraints are implicitly part of labeled feature diagrams. The next theorem shows that constraint may be eliminated completely, resulting in an unlabeled feature diagram.

Theorem 9 (Elimination of Constraints II). *Let E be an arbitrary Boolean expression over a variables' set J. Then, there is an unlabeled* Polynomial *feature diagram over variables set X, $|X| \in \mathcal{O}(|J|)$, and mapping expressions $x = E_x(J)$ for all $x \in X$ and $j = E_j(X)$ for all $j \in J$, which define a bijective mapping between the satisfying assignment of E and the satisfying assignments of F.*

Note that the evolution of 8 into Theorem 9 managed to reduce the number of variables in the feature diagram from exponential to linear; the cost is that variables may occur more than once in the mapping expressions, and in general, these expressions may be exponentially large.

Proof. Instead of giving a full proof, we demonstrate the idea behind it using the same example. Let $F = (x_1 \oplus x_2) \oplus (x_3 \oplus x_4)$ for the expression E of our running example. Then there are precisely 8 satisfying assignments to E, 4 in which precisely one of x_1, x_2, x_3, x_4 is set, and another 4 in which precisely three out of these four variables are set. These 8 assignments define the mappings $i_1 = E_1(x_1, x_2, x_3, x_4), \ldots, i_8 = E_8(x_1, x_2, x_3, x_4)$. We will have that i_k is **1** exactly when x_1, x_2, x_3, x_4 conform to the k^{th} assignment. For example we have $i_1 = x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4$. To obtain the values of the original variables set, we substitute back these expressions into (5), a substitution which brings about the exponential blowup. □

4.3 Equivalence and Inclusion of Polynomial Diagrams

Perhaps the most applicable result we have is that the equivalence and inclusion of Polynomial diagrams can be done in linear time.

Let F_1 and F_2 be labeled Polynomial feature diagrams over the same set of variables V, and let Γ_1 and Γ_2 be (respectively) the set of satisfying assignments to V under F_1 and F_2.

Theorem 10. *There is a linear time algorithm for deciding whether $\Gamma_1 = \Gamma_2$.*

Proof. Since each variable occurs once, the unlabeled version of F_1 and F_2 is a multilinear polynomial of V. Now, apply Shpilka and Volkovich's recent algorithm for identity testing [17], taking care to check for the label variables in each recursive of checking for tree shuffling. □

Theorem 11. *There is a linear time algorithm for deciding whether* $\Gamma_1 \subseteq \Gamma_2$.

Proof. Elementary algebra tells us that the solutions of the polynomial equation $F_1 = \mathbf{1}$ are a subset of the solutions of $F_2 = \mathbf{1}$ whenever the polynomial $F_1 - \mathbf{1}$ divides $F_2 - \mathbf{1}$. Luckily, the Shpilka and Volkovich's algorithm [17] applies also to divisibility. □

5 Related Work and Conclusions

The question of expressiveness of feature diagrams has been discussed in several prior works. Some of these [16,7,19] study this issue via the notion of *Free Feature Diagrams*: a parametrization over the four degrees of freedom that span the space of all feature diagrams.

This paper takes a different approach. Our key observation is that feature diagrams are a special kind of boolean functions: a false decision at any node in the function's expression tree immediately falsifies all variables in the sub-tree rooted at that node.

This observation allows us to abstract over the specifics of feature diagrams language and to rigorously analyze the general concept of "feature diagrams" using the well established techniques of boolean algebra.

Elimination of constraints (via a transformation into a tree) was studied by Broek et-al [20,21]. Those works treat only the most general kind of trees containing all possible operators (mandatory, optional, or and xor). This paper makes the distinction between different kinds of feature diagrams (Sec. 1.2)—based on the mixture of operators used in each kind of diagram—thus making it possible to provide kind-specific results.

Most of the results we present are not encouraging, and while Table 1 enumerates open problems in algorithms and in computational circuit complexity, our conjectures are that no truly efficient algorithms are to pop out in many of those. Nonetheless, as shown in the past [14], theoretical limitations in this area do not necessary impose substantial difficulties in practice.

This work however gave a number of algorithms for efficiently dealing with feature diagrams given in the Polynomial form. It would be interesting to try to implement these, and see whether (i) the implementation of the hard core theoretical computer science algorithms will not introduce intolerable overheads, and whether (ii) the promises of the Polynomial kind of feature diagrams can be fulfilled. Moreover, we believe that these algorithms might be extensible to the more powerful Foda kind, and even deal with arbitrary local constraints of nodes of bounded degree.

Acknowledgments. We thank Amir Shpilka and Ilan Newman for fruitful discussions.

References

1. Alon, N., Boppana, R.B.: The monotone circuit complexity of boolean functions. Combinatorica (1987)
2. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
3. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice (2005)
4. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: SPLC 2007 (2007)
5. Gilbert, E.N.: Lattice theoretic properties of frontal switching functions. J. Math. Phys. (1954)
6. Golumbic, M.C., Mintz, A., Rotics, U.: An improvement on the complexity of factoring read-once boolean functions. Discrete Applied Mathematics (2008)
7. Heymans, P., Schobbens, P.-Y., Trigaux, J.-C., Bontemps, Y., Matulevicius, R., Classen, A.: Evaluating formal properties of feature diagram languages. IET Software (2008)
8. Kaltofen, E.: Factorization of polynomials given by straight-line programs. In: Randomness and Computation, pp. 375–412. JAI Press, Greenwich (1989)
9. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, CMU/SEI-90TR-21 (1990)
10. Karchmer, M., Linial, N., Newman, I., Saks, M., Wigderson, A.: Combinatorial characterization of read once formulae. J. Discrete Math. (1993)
11. Karchmer, M., Wigderson, A.: Monotone circuits for connectivity require superlogarithmic depth. SIAM J. Discrete Math. (1990)
12. McGeer, P.C., Brayton, R.K.: Efficient prime factorization of logic expressions. In: DAC 1989 (1989)
13. Meinel, C., Theobald, T.: Algorithms and Data Structures in VLSI Design. Springer, New York (1998)
14. Mendonca, M., Wasowski, A., Czarnecki, K.: Sat-based analysis of feature models is easy. In: SPLC 2009 (2009)
15. Newman, I., Wigderson, A.: Lower bounds on formula size of boolean functions using hypergraph-entropy. SIAM J. of Discrete Math. (1995)
16. Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks (2007)
17. Shpilka, A., Volkovich, I.: Improved polynomial identity testing of read-once formulas. In: Dinur, I., Jansen, K., Naor, J., Rolim, J. (eds.) Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques. LNCS, vol. 5687, pp. 700–713. Springer, Heidelberg (2009)
18. Thüm, T., Batory, D.S., Kästner, C.: Reasoning about edits to feature models. In: ICSE (2009)
19. Trigaux, J.-C.: Quality of Feature Diagram Languages: Formal Evaluation and Comparison. PhD dissertation, University of Namur, Namur, Belgium (2008)
20. van den Broek, P., Galvão, I.: Analysis of feature models using generalised feature trees. In: Variability Modelling of Software-Intensive Systems (2009)
21. van den Broek, P., Galvão, I., Noppen, J.: Elimination of constraints from feature trees. In: SPLC 2008 (2008)
22. Zave, P.: Feature interactions and formal specifications in telecommunications. Computer (1993)
23. Zhang, W., Zhao, H., Mei, H.: A propositional logic-based method for verification of feature models. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 115–130. Springer, Heidelberg (2004)

Mapping Extended Feature Models to Constraint Logic Programming over Finite Domains

Ahmet Serkan Karataş, Halit Oğuztüzün, and Ali Doğru

Middle East Technical University, Department of Computer Engineering,
06531 Ankara, Turkey
{karatas,oguztuzn,dogru}@ceng.metu.edu.tr

Abstract. As feature models for realistic product families may be quite complicated, automated analysis of feature models is desirable. Although several approaches reported in the literature addressed this issue, complex feature-attribute and attribute-attribute relationships in extended feature models were not handled effectively. In this article, we introduce a mapping from extended feature models to constraint logic programming over finite domains. This mapping is used to translate basic, cardinality-based, and extended feature models, which may include complex feature-feature, feature-attribute and attribute-attribute cross-tree relationships, into constraint logic programs. It thus enables use of off-the-shelf constraint solvers for the automated analysis of extended feature models involving such complex relationships. We also briefly discuss the ramifications of including feature-attribute relationships in operations of analysis. We believe that this proposal will be effective for further leveraging of constraint logic programming for automated analysis of feature models.

Keywords: Variability modeling, extended feature model, feature attribute, constraint logic programming.

1 Introduction

Modeling and managing variability in software product families is a key concept. Among different proposals, feature modeling has been found very effective for managing variability for software product lines (SPLs). Industrial experiences showed that feature models often grow large with hundreds, or even thousands of features and complex cross-tree relationships among features and attributes of features. Therefore automated analysis presents an important challenge for researchers to overcome for adoption of feature modeling for practical product line engineering.

Since their introduction, software researchers have been trying to establish a solid foundation for giving a formal semantics to feature models and reasoning on these models. Theory and implementation behind these foundations have evolved and matured in parallel to the evolving feature model paradigm. But there are still some gaps between the theory and practice that has to be filled. Automated analysis on basic or cardinality-based feature models are covered by some of the earlier studies, but extended feature models where numerical attributes are included lack such in depth coverage [4].

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 286–299, 2010.

As stated by Benavides *et al.* in [4] including feature-attribute relationships for analyses on feature models and proposing new operations of analyses leveraging extended feature models is still a challenge. In this paper we address this challenge and propose a mapping to translate extended feature models including complex feature-feature, feature-attribute and attribute-attribute relationships, to Constraint Logic Programming over Finite Domains, designated CLP(FD). For the sake of completeness we also show that basic and cardinality-based feature models are treated uniformly under our scheme. In addition, we present a brief discussion on including feature-attribute relationships in operations of analyses.

The organization of this paper is as follows. In Section 2, we briefly discuss the feature models and well-known extensions, automated reasoning on feature models, CLP(FD) and point to related work. In Section 3, we present the definitions necessary to form an abstract syntax for our mapping. In Section 4, we propose our approach for the mapping process. In Section 5, we present an illustrative example using an available off-the-shelf CLP(FD) solver. Finally, Section 6 presents discussions and points to future work.

2 Background and Related Work

2.1 Feature Models

A feature is a distinguishable characteristic of a concept (e.g. system, component, etc.) that is relevant to some stakeholder of the concept [18]. A Feature Model is a hierarchically arranged set of features, the relationships defining the composition rules among these features, the relationships defining the cross-tree constraints, and some additional information such as issues/decisions that record various trade-offs, rationale, and justifications for feature selection [14].

Feature models have been popular in Software Product Line Engineering since their introduction by Kang *et al.* as part of Feature Oriented Domain Analysis (FODA) [14]. Following Kang *et al.*'s initial proposal several extensions to feature models have been devised. We refer the reader to [17] for a detailed survey on the feature models. Here we shall briefly discuss the basic feature models and two important extensions to the basic feature models, namely, cardinality-based feature models and extended feature models.

In basic feature models there are two types of relationships among features: decomposition relationships and cross-tree relationships. Decomposition relationships determine the type of the relation between a parent feature and its children, and include four relations: *mandatory*, *optional*, *alternative*, and *or*. Cross-tree relationships are used to specify cross-tree constraints, and include the relationships *requires* and *excludes*. In some feature models a feature diagram is a tree, whereas in some feature models a feature diagram is allowed to be a directed acyclic graph.

An important extension to feature models has been the introduction of cardinalities. In the basic feature models it is possible to express cardinalities, such as 1, 0...1, 0...*, 1...* which covers the most common cases [16]. In practice, however, often situations arise in which a set of features has a multiplicity like 0...6, 1...6, or simply 6.

Cardinality-based feature models include UML-like cardinalities to handle such situations. In [16] Riebisch *et al.* propose to use UML multiplicities as group cardinalities in feature models to be able to express multiplicities like <m..n>. Czarnecki *et al.* propose another cardinality-based extension in [10, 11] that includes feature cardinalities as well as group cardinalities.

Extended feature models provide further information about features using feature attributes. An attribute of a feature is any characteristic of the feature that can be measured. Every feature attribute belongs to a domain, the space of possible values where the attribute takes its values [6]. Domain of an attribute can be discrete (finite or infinite) or continuous; we restrict ourselves to finite domains in the scope of this work.

Kang *et al.* mentioned relationships between features and feature attributes in [14] and "non-functional" features related to feature attributes in [15]. Using attributes in feature models were introduced by Czarnecki *et al.* in [9]. Later Benavides *et al.* [6] proposed a notation for extended feature models.

2.2 Automated Reasoning on Feature Models

Proposals on automated analysis of feature models can be divided into four groups: propositional logic based analyses, description logic based analyses, constraint programming based analyses, and other proposals. Among many automated reasoning analysis approaches of feature models only some of the proposals are capable of handling extended feature models. However, to the best of our knowledge none of the automated reasoning approaches proposed so far cover complex feature-attribute and/or attribute-attribute relationships. We refer the reader to [4] for a detailed literature review on automated analysis of feature models.

Allowing complex feature-feature cross-tree relationships, which are in the form of generic propositional formulas, in feature models were proposed by Batory in his work where he shows the connections among grammars, feature models and propositional formulas [1]. Using constraint programming for automated analyses of feature models were first proposed by Benavides *et al.* [2, 3, 6]. The authors provided a set of mapping rules to translate feature models into a CSP and described how constraint programming based automated reasoning can answer key questions such as *"how many products can be derived from a feature model?"*, *"is it a valid feature model?"*, *"what is the list of all products?"*.

2.3 Constraint Logic Programming over Finite Domains

Constraint Logic Programming (CLP) is essentially a generalization of Logic Programming. In CLP the more general concept of constraint-solving over a computational domain replaces unification, the basic operation of logic programming. In the case of CLP over Finite Domains the domain of each variable is restricted to be finite, often that of finite sets consisting of integer values.

CLP(FD) is regarded a mature field of computer science and proven to be very effective for modeling discrete optimization and verification problems. It also offers a wide variety of analysis facilities, which would be very useful for the implementation of the analysis operations on feature models.

Another important advantage of CLP(FD) is that it is possible to find a variety of off-the-shelf solvers, thus strong tool support is available [12]. The clp(FD) solver [8] we have used to implement our mapping is an example of the solvers for CLP(FD). The clp(FD) solver is available as a library module for SICStus Prolog [13].

3 Abstract Syntax

In this section we present the abstract syntax to construct complex feature-feature, feature-attribute, and attribute-attribute relationships that will serve as the input to the mapping.

Definition 1 *(relop)*: A relop is a relational operator, relop $\in \{=, \neq, <, \leq, >, \geq\}$, where the domains of the operands are restricted to be (possibly infinite) subsets of integers.

Definition 2 *(Compatibility)*: Two domains D_1 and D_2 are *compatible* if and only if $\forall x \in D_1$ and $\forall y \in D_2$, x relop y is well defined.

Example: If $D_1 = \{1, 2, 3\}$, $D_2 = \{1..1000\}$ and $D_3 = \{640x480, 800x600, 1024x768\}$ then D_1 and D_2 are compatible, whereas D_1 and D_3 or D_2 and D_3 are not.

Note that it is always possible to find a mapping from a finite domain to integers. For instance, for the domain D_3 in the previous example one can introduce a mapping such as 640x480 → 1, 800x600 → 2, 1024x768 → 3. We assume that it is the responsibility of the modeler to introduce such mappings as necessary.

Definition 3 *(Condition)*: A *condition* is a Boolean expression either in the form $Expression_1$ *relop* $Expression_2$, or in the form *Feature.attribute* = *truth-value* where;

- Domain of *Feature.attribute* is {true, false}.
- Domains of $Expression_1$ and $Expression_2$ are compatible.
- An expression is
  - i. an integer constant, or
  - ii. value of an attribute, or
  - iii. any well-formed formula constructed by combining integer constants and/or attributes' values with the integer operators {+, –, *, div, mod}.

Example: The following are valid conditions:

- Phone.memory < 256
- X.a ≥ Y.b div 2
- F.a = X.b + Y.c – 60
- Search.case-sensitive = true

One of the strengths of extended feature models is the ability to define complex cross-tree constraints involving feature attributes. For instance *"feature 3D Graphics requires Memory.size ≥ 512"* states that inclusion of the feature *3D Graphics* engenders a constraint on the value of the attribute *size* of the feature *Memory*. To allow inclusion of attributes in the cross-tree relationships in feature models we define cross-tree relationships as follows:

Definition 4 *(Excludes)*: An *excludes* relationship is in the form

P excludes Q

where P and Q are features.

Definition 5 *(Requires)*: A *requires* relationship is in the form

P requires Q

where

- P is a feature.
- Q is
  i. a feature, or
  ii. a condition, or
  iii. any well-formed formula constructed by combining features and/or conditions with the propositional logic connectives.

Example: The following are valid excludes/requires relationships:

- X excludes Y
- X requires Y
- F requires (X.a > 10)
- F requires (X and not Y and (Z.a < X.a or Z.a > 100))

Definition 6 *(Complex Constraint)*: A *complex constraint* is a requires/excludes relationship, or any well-formed formula constructed by combining features and/or requires/excludes relationships with the propositional logic connectives.

Example: The following are valid complex constraints:

- A and B implies not C
- F requires (X.a > Y.b or Z.c > 100)
- (X excludes Y) or (X excludes Z)
- (F requires ((X.a > 256 and Z) or (X.a > 512))) and (F excludes Z)

We have allowed features to take part in complex constraints so that our definition agrees with the mapping of complex feature-feature cross-tree relations described by Batory [1].

In some cases it may be desirable to specify conditional constraints as well. For instance, *"if the value of the attribute maximum number of participants of the feature Conference Call is greater than 4, then the feature Conference Call requires Connection.speed ≥ 1024 kbps"* is a conditional constraint. If the value of the *maximum number of participants* is not greater than 4 then the feature *Conference Call* imposes no constraints on the attribute *speed* of the feature *Connection*. The constraint becomes effective if the condition is true. To allow such cases we introduce the concept of *guarded constraint*.

Definition 7 *(Guarded Constraint)*: A guarded constraint is a relationship in the form

If *Guard* then *Complex Constraint*

where *Guard* is any Boolean combination of conditions.

As in the case of complex constraints, guarded constraints can be combined with the propositional logic connectives to build more complex constraints.

Features and attributes of features are at two different levels of detail. In order to treat these two levels uniformly in a mapping we introduce the notion of an *implicit attribute*.

Definition 8 *(Implicit Attribute)*: Every feature has an *implicit attribute* named *selected* that ranges over the domain {true, false}.

An implicit attribute gets the value true if the feature is selected to be a part of some product and false otherwise. Therefore the value of the attribute *selected* of the concept feature (the root of the feature diagram) will always be true.

Example: If X.selected = true, then the feature X is a part of the product, whereas if Y.selected = false, then feature Y is not.

4 Mapping

In this section we describe our mapping approach. The mapping from a feature model to a CLP(FD) program has the following properties:

- The attributes, including the implicit attributes *'selected'* of each feature, make up the set of variables.
- The domains of all attributes are finite. The domain of the implicit attribute *'selected'* of each feature is *{true, false}*.
- Every relationship (decomposition, cross-tree) becomes a constraint.

First we present the mapping for decomposition relationships in feature models in Section 4.1, and then we present the mapping for complex feature-feature, feature-attribute, and attribute-attribute cross-tree relationships in Section 4.2.

4.1 Mapping Decomposition Relationships

Mapping Mandatory Relation: Let P and C be two features, where P is the parent of C, in a *mandatory relation*, then the equivalent constraint is: $P.selected \Leftrightarrow C.selected$

Mapping Optional Relation: Let P and C be two features, where P is the parent of C, in an *optional relation*, then the equivalent constraint is: $C.selected \Rightarrow P.selected$

Mapping Or Relation: Let P, C_1, C_2, …, and C_n be features, where P is the parent of C_1, C_2, …, and C_n, in an *or relation*, then the equivalent constraint is:
$C_1.selected \vee C_2.selected \vee \ldots \vee C_n.selected \Leftrightarrow P.selected$

Mapping Alternative Relation: Let P, C_1, C_2, …, and C_n be features, where P is the parent of C_1, C_2, …, and C_n, in an *alternative relation*, then the equivalent constraint is:
$(C_1.selected \Leftrightarrow (\neg C_2.selected \wedge \ldots \wedge \neg C_n.selected \wedge P.selected)) \wedge \ldots \wedge$
$(C_n.selected \Leftrightarrow (\neg C_1.selected \wedge \ldots \wedge \neg C_{n-1}.selected \wedge P.selected))$

Mapping Feature Cardinality: Let F be a feature with the cardinality <*n*, *m*>, then we map the feature as an optional feature if $n = 0$, and as a mandatory feature otherwise. If the upper bound is a constant (e.g. not *) then we also propose to map an extra attribute for feature F with the name *number of instances* and the domain {*n*, …, *m*} as described in the following subsection. This additional mapping enables including the number of instances of this feature in the complex cross-tree constraints.

Mapping Group Cardinality: Let P, C_1, C_2, …, and C_k be features, where P is the parent of C_1, C_2, …, and C_k, and <*n*, *m*> the group cardinality in a decomposition with *group cardinality*. To give our mapping we make use of the *choose* operator proposed by Batory in [1]. In general, $choose_{n,m}(e_1 \ldots e_k)$ means that at least *n* and at most *m* of the expressions $e_1 \ldots e_k$ are true, where $0 \leq n \leq m \leq k$. Then the equivalent constraint is: $choose_{n,m}(C_1.selected, \ldots, C_k.selected) \Leftrightarrow P.selected$

A possible implementation of the $choose_{n,m}$ operator would be as follows. Let S be the set of subsets of $C = \{C_1, C_2, \ldots, C_k\}$ with cardinality at least *n* and at most *m*, then we construct the following subformula for every $S_i \in S$ and $T_i = C - S_i$, such that $S_i = \{X_1, X_2, \ldots, X_p\}$ and $T_i = \{Y_1, Y_2, \ldots, Y_q\}$:

$$f_i = (X_1.selected \wedge \ldots \wedge X_p.selected \wedge \neg Y_1.selected \wedge \ldots \wedge \neg Y_q.selected)$$

Thus, we will have $|S|$ subformulas: $f_1, f_2, \ldots, f_{|S|}$. Then, we combine all these subformulas with logical disjunction:

$$F = f_1 \vee f_2 \vee \ldots \vee f_{|S|}$$

Finally, the mapping is as follows:

$$(P.selected \Rightarrow F) \wedge (\neg P.selected \Rightarrow (\neg C_1.selected \wedge \ldots \wedge \neg C_n.selected))$$

The reader would have noticed that we have adopted an approach similar to the one proposed by Benavides *et al.* [2, 6] while mapping decomposition relationships. The main difference is that features make up the variables in the proposal of Benavides *et al.* whereas attributes make up the variables in our proposal. Note that this part is not the original part of our proposal, but we have included it to show that basic and cardinality-based decomposition relationships in feature models are treated and mapped to CLP(FD) uniformly under our scheme. Therefore this part, combined with the mapping presented in the following subsection, establishes the completeness of our proposal.

Mapping Complex Cross-Tree Relationships

Mapping Domains of Attributes: Let F be a feature and *a* an attribute of F with the domain $D_a = \{n_1, \ldots, n_k\}$, then the equivalent constraint is: $F.a \in D_a$

Mapping a Condition: Let C be a *condition*, and $\{F_1.a_1, \ldots, F_1.a_i, \ldots, F_n.b_1, \ldots, F_n.b_k\}$ the set of all attributes except the implicit attributes involved in C, then the equivalent constraint is: $(C \wedge F_1.selected \wedge \ldots \wedge F_n.selected)$

Mapping Excludes Relationship: Let X and Y be the features in an *excludes* relationship such that *X excludes Y*, then the equivalent constraint is:
$\neg$ (X.selected $\wedge$ Y.selected)

Mapping Requires Relationship: Let F be the feature and E the expression in a *requires* relationship such that *F requires E*, then we first build the equivalent constraint for the expression E *(ECE)* in two steps, where the ordering of the steps is not important, as follows:

- Let $\{F_1, \ldots, F_n\}$ be the features in E, then each F_i, where $i \in \{1, 2, \ldots, n\}$, is replaced with F_i.selected.
- Let $\{C_1, \ldots, C_m\}$ be the conditions in E, then each C_i, where $i \in \{1, 2, \ldots, m\}$, is replaced with the ECC_i, where ECC_i is the equivalent constraint for C_i.

Then the equivalent constraint for the *requires* relationship is: F.selected ⇒ ECE

Example: Consider the requires relationship *F requires (X and (Z.a < X.a or Z.a > 100))*. When we replace each feature in the expression with its implicit attribute we get the following formula:

$$X.selected \wedge (Z.a < X.a \vee Z.a > 100)$$

In this formula we have two conditions; *(i) Z.a < X.a* and *(ii) Z.a > 100*, where these conditions are mapped as:

i. $(Z.a < X.a \wedge Z.selected \wedge X.selected)$
ii. $(Z.a > 100 \wedge Z.selected)$

Thus, we get the following *ECE*:

$$X.selected \wedge ((Z.a < X.a \wedge Z.selected \wedge X.selected) \vee (Z.a > 100 \wedge Z.selected))$$

Finally, the equivalent constraint for the sample requires relationship is:

F.selected ⇒

$$X.selected \wedge ((Z.a < X.a \wedge Z.selected \wedge X.selected) \vee (Z.a > 100 \wedge Z.selected))$$

Mapping a Complex Constraint: Let $\mathcal{C}$ be a complex constraint, $\{R_1, \ldots, R_n\}$ the set of requires relationships in $\mathcal{C}$, $\{E_1, \ldots, E_m\}$ the set of excludes relationships in $\mathcal{C}$, and $\{F_1, \ldots, F_k\}$ the set of features that are not part of an excludes or requires relationship in $\mathcal{C}$. Then the equivalent constraint for $\mathcal{C}$ is built in three steps, where the ordering of the steps is not important, as follows:

- Replace each R_i in $\mathcal{C}$ with its equivalent constraint where $i \in \{1, 2, \ldots, n\}$.
- Replace each E_i in $\mathcal{C}$ with its equivalent constraint where $i \in \{1, 2, \ldots, m\}$.
- Replace each F_i in $\mathcal{C}$ with F_i.selected where $i \in \{1, 2, \ldots, k\}$.

Example: Consider the complex constraint *(X excludes Y) or (F requires Z.a > 10)*. The equivalent constraint for the excludes relationship is:

$$\neg (X.selected \wedge Y.selected)$$

The equivalent constraint for the requires relationship is:

$$F.selected \Rightarrow (Z.a > 10 \wedge Z.selected)$$

When we replace these equivalent constraints we get the following final constraint:

$$\neg (X.selected \wedge Y.selected) \vee (F.selected \Rightarrow (Z.a > 10 \wedge Z.selected))$$

Mapping a Guarded Constraint: Let $\mathcal{GC}$ be a guarded constraint such that *if* $\mathcal{G}$ *then* $\mathcal{CC}$, then we first build the equivalent constraints for the guard $\mathcal{G}$ and the complex constraint $\mathcal{CC}$ as follows:

- Let $\{C_1, \ldots, C_n\}$ be the conditions in $\mathcal{G}$. Then, each C_i, where $i \in \{1, 2, \ldots, n\}$, is replaced with the ECC_i, where ECC_i is the equivalent constraint for C_i, which gives us the ECG *(Equivalent Constraint for* $\mathcal{G}$*)*.
- The ECCC *(Equivalent Constraint for* $\mathcal{CC}$*)* is built as shown above.

Then the equivalent constraint for the guarded constraint is: $\mathrm{ECG} \Rightarrow \mathrm{ECCC}$

Example: Consider the guarded constraint *if X.a > 10 and Y.selected = true then F requires Z.b = 20*. To build the *ECG* we replace X.a > 10 with its ECC $\mathrm{X.a} > 10 \wedge \mathrm{X.selected}$, which gives us:

$$(\mathrm{X.a} > 10 \wedge \mathrm{X.selected}) \wedge \mathrm{Y.selected} = \mathrm{true}$$

The ECCC, equivalent constraint for *F requires Z.b = 20*, is:

$$\mathrm{F.selected} \Rightarrow (\mathrm{Z.b} = 20 \wedge \mathrm{Z.selected})$$

Thus, the sample guarded constraint is mapped as:

$$((\mathrm{X.a} > 10 \wedge \mathrm{X.selected}) \wedge \mathrm{Y.selected} = \mathrm{true}) \Rightarrow (\mathrm{F.selected} \Rightarrow (\mathrm{Z.b} = 20 \wedge \mathrm{Z.selected}))$$

When all the mappings are completed, the formulas thus obtained are combined with the logical conjunction, which yields the corresponding formula for the whole feature model.

5 An Example Mapping

In this section we present an example mapping from the feature model given in Figure 1 to CLP(FD). To implement this example we have used the clp(FD) solver [8].

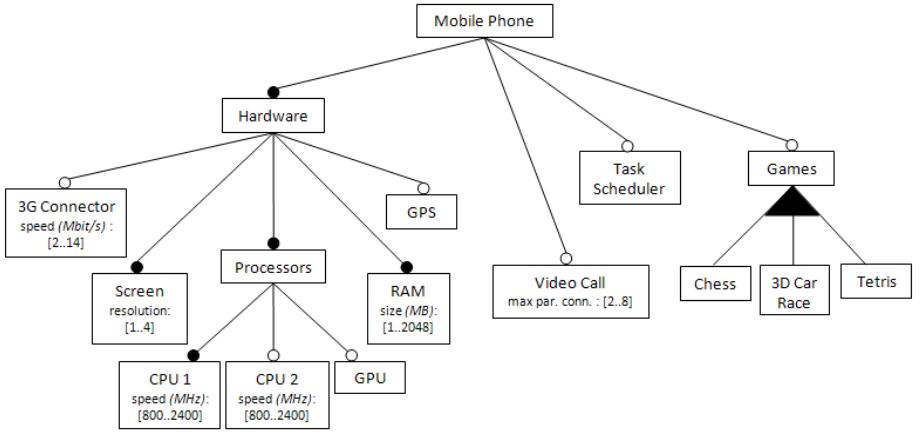


Fig. 1. A Sample Feature Model for a Mobile Phone

Assume that we have the following complex cross-tree relationships in the sample feature model:

- (C1) When there are two CPU's on board, Task Scheduler must be a part of the product (*CPU 1* and *CPU 2* implies *Task Scheduler*)
- (C2) If *Video Call.mpc* ≥ 4 then *Video Call* requires *Screen.resolution* ≥ 320x640 and *3G Connector.speed* ≥ 6
- (C3) *3D Car Race* requires (*GPU* and *RAM.size* ≥ 512) or (*RAM.size* ≥ 1024)
- (C4) If *Screen.resolution* < 320x640 then *Screen* excludes *GPS*
- (C5) *Task Scheduler* requires *CPU 1.speed* ≥ *CPU 2.speed*

Note that we have an attribute, namely *Screen.resolution*, with the domain {128x160, 240x320, 320x640, 360x640} which does not consist of integer values. As we allow only integer values in operands of a relop we introduce the following conversion before we start: {128x160 → 1, 240x320 → 2, 320x640 → 3, 360x640 → 4}, thus the domain of the attribute *Screen.resolution* becomes {1, 2, 3, 4}.

First, we map the domains of the *implicit attributes* to their equivalent constraints. All of the implicit attributes will range over the domain {true, false}.

MobilePhone.selected ∈ {true, false} ∧ Hardware.selected ∈ {true, false} ∧
TaskScheduler.selected ∈ {true, false} ∧ Games.selected ∈ {true, false} ∧
... ∧ Tetris.selected ∈ {true, false}

Next, we map the domains of the other attributes to their equivalent constraints.

ThreeGConn.speed ∈ {2, ..., 14} ∧ Screen.resolution ∈ {1, ..., 4} ∧
RAM.size ∈ {1, ..., 2048} ∧ VideoCall.mpc ∈ {2, ..., 8} ∧
CPU1.speed ∈ {800, ..., 2400} ∧ CPU2.speed ∈ {800, ..., 2400}

After mapping of the domains is complete we start mapping the decomposition relationships to their equivalent constraints, and start with mapping the *mandatory relations*.

MobilePhone.selected ⇔ Hardware.selected ∧
Hardware.selected ⇔ Screen.selected ∧
Hardware.selected ⇔ Processors.selected ∧
Hardware.selected ⇔ RAM.selected ∧
Processors.selected ⇔ CPU1.selected

Then, we map the *optional relations*.

VideoCall.selected ⇒ MobilePhone.selected ∧
Games.selected ⇒ MobilePhone.selected ∧
...
∧
GPU.selected ⇒ Processors.selected

Next, we map the *or relation*.

(Chess.selected ∨ ThreeDCRace.selected ∨ Tetris.selected) ⇔ Games.selected

At this point we have finished mapping decomposition relations for our example, so we start mapping the complex cross-tree relations. The first constraint we will map is C1, which is a *complex constraint*.

$$(\text{CPU1.selected} \wedge \text{CPU2.selected} \Rightarrow \text{TaskScheduler.selected})$$

Next, we map C2, which is a *guarded constraint*.

$$(\text{VideoCall.mpc} \geq 4 \wedge \text{VideoCall.selected}) \Rightarrow (\text{VideoCall.selected} \Rightarrow \text{Screen.resolution} \geq 3 \wedge \text{Screen.selected} \wedge \text{ThreeGConn.speed} \geq 6 \wedge \text{ThreeGConn.selected})$$

Then, we map C3.

$$\text{ThreeDCRace.selected} \Rightarrow (\text{GPU.selected} \wedge \text{RAM.size} \geq 512 \wedge \text{RAM.selected}) \vee (\text{RAM.size} \geq 1024 \wedge \text{RAM.selected})$$

Next, we map C4, another *guarded constraint*.

$$(\text{Screen.resolution} < 3 \wedge \text{Screen.selected}) \Rightarrow \neg\, (\text{Screen.selected} \wedge \text{GPS.selected})$$

Last constraint to map is C5, which includes an attribute-attribute relation.

$$\text{TaskScheduler.selected} \Rightarrow (\text{CPU1.speed} \geq \text{CPU2.speed} \wedge \text{CPU1.selected} \wedge \text{CPU2.selected})$$

Finally we combine all of the formula above with the logical conjunction to get the complete formula for the whole model.

Using the clp(FD) solver, with the above formula coded, some of the products automatically derived by the solver are as follows:

- {Mobile Phone, Hardware, Screen, Screen.resolution ∈ {1, …, 4}, Processors, CPU 1, CPU 1.speed ∈ {800, …, 2400}, RAM, RAM.size ∈ {1, …, 2048}}
- {Mobile Phone, Hardware, Screen, Screen.resolution ∈ {3, 4}, Processors, CPU 1, CPU 1.speed ∈ {800, …, 2400}, CPU 2, CPU 2.speed ∈ {800, …, 2400}, RAM, RAM.size ∈ {1024, …, 2048}, GPS, Task Scheduler, Games, Chess, 3D Car Race}
- {Mobile Phone, Hardware, Screen, Screen.resolution ∈ {3, 4}, Processors, CPU 1, CPU 1.speed ∈ {800, …, 2400}, CPU 2, CPU 2.speed ∈ {800, …, 2400}, GPU, RAM, RAM.size ∈ {512, …, 2048}, GPS, 3G Connector, 3G Connector.speed ∈ {6, …, 14}, Task Scheduler, Games, Chess, 3D Car Race, Video Call, Video Call.mpc ∈ {4, …, 8}}

6 Discussions and Future Work

In this paper we have proposed a mapping from extended feature models, which may include complex feature-feature, feature-attribute and attribute-attribute cross-tree relationships, to constraint logic programming over finite domains. For the sake of

completeness we have also shown that basic and cardinality-based feature models are treated uniformly under our scheme. The only restriction we impose is that the domains of the attributes be finite.

One of the strengths of this proposal originates from the availability of a wide variety of off-the-shelf CLP(FD) solvers being used for many real-life applications. CLP(FD) systems evolved to provide efficient implementations for the computationally expensive procedures that had proven to be very effective for modeling discrete optimization and verification problems. Moreover, due to their declarative style, CLP systems lead to highly readable solutions. The tools also provide a wide variety of facilities for the users [12].

Therefore, once the mapping is completed it is straightforward to implement many useful analysis operations, such as *model validation*, *product checking*, *valid partial configuration checking*, *finding all products, calculating number of products, calculating variability, calculating commonality, obtaining commonality of a feature*, *optimization* and so on. In particular, CLP(FD) systems provide various higher-order predicates, such as *minimize(Goal, Function)* and *minimize-maximum(Goal, [F_1, …, F_n])*, that can be used for *optimization* operations. One can also make use of the wide variety of built-in predicates available for list manipulation to perform operations such as *finding all products*, *calculating number of products*, or *calculating variability*. (The reader may consult [4] for the definitions of the aforementioned operations.) The efficiency of these implementations in the view of the demands of product line engineering practice remains to be appraised.

We should note that, the inclusion of feature-attribute relationships would require revised definitions of some of the analysis operations. For instance, consider the simple extended feature model in Figure 2.

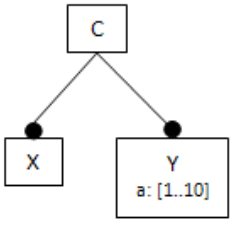


Fig. 2. A Simple Extended Feature Model

Assume that there is a cross-tree relationship, *"X requires Y.a > 6"*. When we seek the answer to the question *"which are the products of this model?"* there might be two possible ways to answer.

The first answer would include a single product:

P1 = {C, X, Y where Y.a ∈ {7, 8, 9, 10}}

Whereas the second answer would list 4 products:

P1 = {C, X, Y where Y.a = 7} through P4 = {C, X, Y where Y.a = 10}

A feature has two possible states: it is either a part of a product, or not. The situation, however, may not be so simple when attributes are involved. An attribute may take on a variety of values once the feature it belongs to is a part of the product. This brings about a new kind of variability: *variability in terms of attributes*.

Partially specialized features, the features that still have unresolved variability in their attributes' values, bring into discussion the concept of *feature specialization* and *feature configurations*. Thus, it would be necessary to introduce *configurations of features*, which can also be used during staged configurations using feature models [10]. Introduction of a new type of variability, such as *variability in the value of the attributes of a feature*, would require the reformulation of analysis operations such as "number of products", "all products", "filter" and so on. For instance, as there is a new level of variability, calculations for the variability factor of a feature model and commonality of a configuration with respect to a feature model would have to be reformulated. Clearly, this subject needs further research and rigorous examination of the effects of including feature-attribute and attribute-attribute relationships for analyses on feature models.

Another challenge, regarding the inclusion of feature-attribute and attribute-attribute relationships for the analyses of feature models, is the introduction of new operations of analysis [4]. Industrial experience would provide motivation on this matter, but it may also be possible to discover new requirements arising from the nature of the analysis needs. Further research is needed on this issue as well.

As the mapping from extended feature models to CLP(FD) has a well-defined structure, it would be possible to incorporate it into existing feature modeling and analysis tools such as FAMA [5] or develop new tools. Such an effort would require the design of a formal language to represent extended feature models that may include complex feature-feature, feature-attribute and attribute-attribute cross-tree relationships, or extend existing languages such as TVL [7] as necessary.

Our ongoing research activities focus on three issues: *(i)* the need for the reformulation of some or all of the existing operations of analysis, *(ii)* the need for introduction of new operations of analysis, *(iii)* development of a tool (possibly built upon free software) that will automate the mapping and analysis operations.

References

1. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
2. Benavides, D., Ruiz-Cortés, A., Trinidad, P.: Coping with automatic reasoning on software product lines. In: Proceedings of the 2nd Groningen Workshop on Software Variability Management (November 2004)
3. Benavides, D., Ruiz-Cortés, A., Trinidad, P.: Using constraint programming to reason on feature models. In: The Seventeenth International Conference on Software Engineering and Knowledge Engineering, SEKE 2005, pp. 677–682 (2005)

4. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: A literature review. Information Systems 35(6), 615–636 (2010)
5. Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortés, A.: FAMA: Tooling a framework for the automated analysis of feature models. In: Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS), pp. 129–134 (2007)
6. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated Reasoning on Feature Models. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg (2005)
7. Boucher, Q., Classen, A., Faber, P., Heymans, P.: Introducing TVL, a Text-based Feature Modeling Language. In: Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010), Linz, Austria, January 27-29, pp. 159–162 (2010)
8. Carlsson, M., Ottosson, G., Carlson, B.: An Open-Ended Finite Domain Constraint Solver. In: Proc. Programming Languages: Implementations, Logics, and Programs (1997)
9. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.: Generative programming for embedded software: An industrial experience report. In: Proceedings of the ACM SIGPLAN/ SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002). LNCS, vol. 2487, pp. 156–172. Springer, Heidelberg (2002)
10. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configurations Using Feature Models. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 266–283. Springer, Heidelberg (2004)
11. Czarnecki, K., Kim, C.H.P.: Cardinality-based feature modeling and constraints: a progress report. In: International Workshop on Software Factories, San Diego, California (October 2005)
12. Fernandez, A., Hill, P.M.: A comparative study of eight constraint programming languages over the Boolean and finite domains. Journal of Constraints 5, 275–301 (2000)
13. `http://www.sics.se/isl/sicstuswww/site/index.html`
14. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analyses (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh (1990)
15. Kang, K., Kim, S., Lee, J., Kim, K.: FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering 5, 143–168 (1998)
16. Riebisch, M., Bollert, K., Streitferdt, D., Philippow, I.: Extending Feature Diagrams With UML Multiplicities. In: 6th Conference on Integrated Design & Process Technology (IDPT 2002), Pasadena, California, USA (2002)
17. Schobbens, P., Trigaux, J.C., Heymans, P., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks 51(2), 456–479 (2007)
18. Simos, M., et al.: Software Technology for Adaptable Reliable Systems (STARS) Organization Domain Modeling (ODM) Guidebook Version 2.0., STARS-VC-A025/001/00, Manassas, VA, Lockheed Martin Tactical Defense Systems (1996)

Stratified Analytic Hierarchy Process: Prioritization and Selection of Software Features

Ebrahim Bagheri[1,3], Mohsen Asadi[1,2], Dragan Gasevic[1], and Samaneh Soltani[1]

[1] Athabasca University, Canada
[2] Simon Fraser University, Canada
[3] National Research Council Canada
{ebagheri,dragang}@athabascau.ca, masadi@sfu.ca,
soltanisa@gmail.com

Abstract. Product line engineering allows for the rapid development of variants of a domain specific application by using a common set of reusable assets often known as *core assets*. Variability modeling is a critical issue in product line engineering, where the use of feature modeling is one of most commonly used formalisms. To support an effective and automated derivation of concrete products for a product family, staged configuration has been proposed in the research literature. In this paper, we propose the integration of well-known requirements engineering principles into stage configuration. Being inspired by the well-established Preview requirements engineering framework, we initially propose an extension of feature models with capabilities for capturing business oriented requirements. This representation enables a more effective capturing of stakeholders' preferences over the business requirements and objectives (e.g.,. implementation costs or security) in the form of fuzzy linguistic variables (e.g., high, medium, and low). On top of this extension, we propose a novel method, the Stratified Analytic Hierarchy process, which first helps to rank and select the most relevant high level business objectives for the target stakeholders (e.g., security over implementation costs), and then helps to rank and select the most relevant features from the feature model to be used as the starting point in the staged configuration process. Besides a complete formalization of the process, we define the place of our proposal in existing software product line lifecycles as well as demonstrate the use of our proposal on the widely-used e-Shop case study. Finally, we report on the results of our user study, which indicates a high appreciation of the proposed method by the participating industrial software developers. The tool support for S-AHP is also introduced.

1 Introduction

A key aspect of software product line engineering is capturing the common characteristics of a set of software-intensive applications in a specific problem domain [1]. Product line engineering allows for the rapid development of variants of a domain specific application by using a common set of reusable assets often known as *core assets*. Such an approach supports the management of commonality as well as variability in the software development process [2][3]. Feature modeling is an important conceptual tool that offers modeling support for software product lines. It provides for addressing commonality and variability both formally and graphically, describing interdependencies of the product family attributes (features) and expressing the permissible variants and configurations of the product family.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 300–315, 2010.

One of the important issues in any of the feature modeling methodologies is the selection of the best and at the same time allowable combination of features that would satisfy the mission of the target application and the stakeholders' requirements. Many researchers in the software product line domain have proposed techniques to deal with these challenges and produce appropriate software product line configurations [4][5][6]. In order to handle a large number of comparisons during the configuration process, in many cases, a configuration is gradually developed in several stages breaking down a large amount of required feature comparisons into a set of consecutive stages. In each stage, a subset of preferred features are selected and finalized and the unnecessary features are discarded yielding a new feature model whose set of possible configurations are a subset of the original feature model. This feature model is referred to as a *specialization* of the feature model and the staged refinement process constitutes *staged configuration* [6].

For an effective staged configuration process there is a need to systematically manage different business-oriented requirements, which will drive the process of staged configuration. This can be addressed by extending feature modeling languages with attributes reflecting business-specific concerns (e.g., security and implementation costs). In this paper, we propose such an extension to cardinality-based feature models in Section 2. While this expressivity might appear quite useful, when it is combined with feature models, it can sometimes be a double-edged sword. That is, such comprehensiveness makes the selection of the best set of features for a particular application difficult. Therefore, it is important to understand the relative importance of features and business-specific concerns of a product family from various perspectives. The difference in the importance and priority of features in dissimilar domains would allow for the creation of application specific preference ordering between them.

It is clear that for an advanced staged configuration process, in addition to the use of feature models, there is a need to both capture and leverage business-oriented requirements more effectively. Such a support method must enjoy three important characteristics to be of interest to the professional software engineers: *i) It must be easy, straightforward and fast to use*: Complicated methods that take too many factors into account and are hence slow often fail to become prevalent in industrial settings; *ii) It should provide correct, reliable and provable/repeatable outcomes*: Provability is an important factor in any software engineering process, since the underlying reasons for certain decisions need to be traceable back to their original causes and roots for future decision making purposes; iii) *It needs to be able to effectively communicate with and capture the intentions of the process stakeholders*: This is a very important success factor as stakeholders' satisfaction plays a key role in the continuity of the project.

In this paper, we introduce a new method called the Stratified Analytic Hierarchy Process (S-AHP) for prioritizing (ranking) and filtering the features of a product family to enhance and expedite the feature selection and product configuration process (Section 3). S-AHP is basically concerned with finding the most appropriate set of features that need to be included in the final software product, given the stakeholders' requirements and their important goals and objectives. The output of S-AHP will be the input of typical feature modeling configuration algorithms that specialize a feature model based on the stakeholders' requests and relevant integrity constraints. S-AHP is based on a pair-wise comparison scheme, which although is simple to use and

straight-forward to comprehend for software practitioners, it can at the same time significantly reduce the number of required comparisons from an otherwise computationally explosive number of possibilities. Furthermore, it provides means for tracing between stakeholders' objectives and the feature selection decisions. To support for the systematic use of the proposed method, we show how the proposed method can be incorporated into the current SPL lifecycles and outline the tooling support that we have provided for our method (Section 4). Further, we describe a detailed case study illustrating the proposed method (Section 5). Before concluding the paper, the results of a user study that we conducted to evaluate S-AHP will be provided (Section 6) and relevant related work will be reviewed (Section 7).

2 Extending Cardinality-Based Feature Models

The cardinality-based notation for feature modeling integrates four existing extensions of the FODA notation: feature cardinalities, group cardinalities, feature diagram references, and attributes [2]. Cardinality-based feature models are based on a metamodel that defines their abstract syntax [6]. In Figure 1, we depict a modified version of its metamodel where some important concepts for the purpose of our work have been included and two new concepts are added to the meta-model. As it can be seen in the extended meta-model, each feature can be annotated with one or more *concerns*. We have adopted the concept of concerns from the Preview framework in the multiple-viewpoint requirements elicitation domain [7]. The Preview framework has improved the requirements elicitation process by the explicit identification of the importance of organizational needs and priorities via the recognizing of *concerns*. The role of concerns in Preview is to concentrate on factors that are central to the success of a system under development. Similarly we employ concerns within the feature selection process to understand and evaluate the factors that are central to the final product's success. Therefore, in order to enhance the selection of features for a particular application, during the feature modeling process we should take the concerns of the business requirements and high-level objectives of the stakeholders and their relative importance and priorities into account. Similar to Preview, used for requirements analysis of a single system, we consider concerns as the high-level strategic objectives of the application domain and product family stakeholders. Hence, they can be used to ensure consistency and alignment between the vital goals pursued by the design of the product family and the product family features.

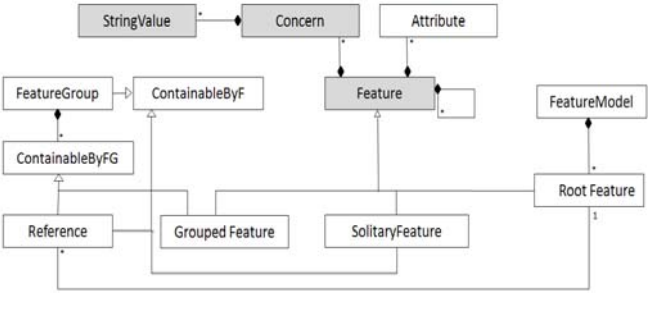


Fig. 1. A modified meta-model for the cardinality-based feature models

Concerns should be expressed at a high-level of abstraction to avoid overlap with some of the actual features of the product family. Examples of concerns can include but are not limited to *implementation cost, time, risk, volatility, customer importance,* and *penalty*. Furthermore, each concern can be described by a textual description tag, referred to in this paper as a *qualifier tag*. For instance, options available for cost could be *expensive*, *affordable*, and *cheap*. In this way, each feature will be assigned several concerns based on the discretion of the modeler. Each concern can be further described by a tag qualifying that concern. A good choice for qualifier tags would be the use of fuzzy linguistic variables such as high, medium, and low. This will give the modeler the option to qualify assigned tags for different concerns in a similar way.

3 Business Centered Staged Configuration

In this section, we introduce our proposed method (S-AHP) that satisfies the requirements mentioned in the introduction for an efficient feature selection process and considers the stakeholders' concerns, goals and strategic objectives during its course. Our proposal is based on the Analytic Hierarchy Process (AHP) [8], which is a well-established framework in the domain of decision theory.

3.1 Analytic Hierarchy Process

AHP [8] is a rather easy-to-use pair-wise comparison method that computes relative rankings of various phenomena based on the judgments of its users. Compared with methods that are based on absolute value assignment such as scale-based (1-to-10) rankings and voting schemes [1], AHP is less susceptible to judgmental errors and inaccuracy [9]. Furthermore, AHP provides complete justification for its ranking results based on the provided comparisons. AHP undertakes a pair-wise comparison process between the objectives of a study. AHP takes a matrix in which each row of the matrix contains observations of the stakeholders' judgments on the relative importance of a phenomenon compared to some other phenomena. Given this matrix, AHP computes the relative importance and ranking of the available phenomena. The steps to perform AHP are summarized as follows:

1. Let $M\ (n, n)$ be an input square matrix and $m_{i,*}$ refers to the i^{th} row of M and $m_{*,j}$ refers to the j^{th} column of M. Each cell in the matrix such as $m_{i,j}$ represents the relative comparison value between the phenomena i and j;
2. The columns of M are normalized in such a way that each cell is divided by the sum of the values in its column;
3. The eigenvalues of the normalized matrix are calculated;
4. The relative importance and rank of $m_{i,*}$ is computed as $R(m_{i,*}) = (\Sigma_{0<j<|n|} M[i, j])/n$.

By performing the above steps, AHP computes a rank for each row, where each row corresponds to some phenomenon. Suppose AHP is employed to rank the n features present in a given feature model. AHP will take a square matrix with size of n as a starting point. Each cell in the matrix will contain a relative importance of a feature regarding the other features. Therefore, in order to compute the relative ranking of

each feature, AHP will need to perform $n \times (n-1)/2$ comparisons. Since complicated feature models have a large size and some features are not comparable with the others, it would not be suitable to use AHP directly for feature ranking. In our proposed S-AHP method, we employ concerns attached to the features to reduce the number of required comparisons and to make the comparison of heterogeneous features feasible.

3.2 Formalizing the Configuration Process

Generally, the feature model and its annotated information (i.e. concerns and concerns tag values) in our context can be formalized as a triple $<F, C, V>$ where

- F is the initial feature model;
- C is the set of concerns relevant for the feature model (e.g. implementation cost and security);
- V is the set of concern tag values (e.g. high, low, medium)

Based on this triple, we define the feature configuration process as follow:

$$< F, C, V > \xrightarrow{\text{Concern prioritization stage}} < F', C', V' > \xrightarrow{\text{feature ranking stage}} < F', C', V' > \xrightarrow{\text{Feature specialization stage}} < F'', C', V' >$$

The specialization process starts given the initial feature model, the concern set, and the set of tag values relevant to each concern. First, the concern prioritization stage is performed by executing AHP to identify the desirable set of business concerns (C′) from the stakeholders' perspective. This stage gathers the relative importance of each concern in comparison to the other concerns from the involved stakeholders. This would allow us to filter out the least significant concerns and their corresponding tag values from V and a new set V' is created. Moreover, features whose concern set are a subset of the removed concerns, and their deletion does not violate the feature model constraints are removed from the feature model, which entails the creation of a new feature model F'. Then, the relative importance of the concern tags is calculated. This process is done by employing the AHP algorithm with the matrix containing the values of relative importance for each concern tag. Finally, the calculated rank for each of the tag values are applied to the relevant features. In cases when a feature has been annotated with more than one concern, the representative rank value of that feature is selected based on the users' opinion. Finally, the feature specialization step is done iteratively to select or remove features from the feature model (F') based on their ranks and constraints.

3.3 Feature Configuration Stages

As it can be seen from the process explained above, ranking of features is performed in two layers, which create a stratified (layered) process: i) the identified concerns are prioritized and the less important concerns are filtered out; and ii) the qualifier tags of the remaining selected concerns are compared and prioritized. The structure and formulation of S-AHP is such that it significantly reduces the number of comparisons re-

quired for ordering the features of a given feature model. This section describes the stages performed to produce a valid feature model configuration with S-AHP.

3.3.1 Concern Prioritization Stage. This stage aims at comparing and ranking the list of concerns for a specific target application. All defined concerns on the feature model and their relative importance for the target application are considered, and the pair-wise comparison process (AHP) is undertaken to produce a ranked list of concerns. It can be realized as follows:

1. Assume C is set of concerns. A square matrix $P_{|C|\times|C|}$= *{P[i, j] = α | 1 ≤ i, j ≤ |C| and α is relative importance of concern i to concern j}* is created. Traditionally within AHP, the values 1, 3, 5, 7, and 9 have been used to represent the degree of importance showing *equal value, slightly more value, strong valued, very strong value* and *extreme value*, respectively; Each cell in the P such as $p_{i,j}$ contains one of the values from 1, 3, 5, 7 and 9 and shows how significant concern *i* is with respect to concern *j;*
2. The steps of AHP are performed on matrix *P* where $P_{i,*}$ (i.e., row *i* of matrix *P*) shows the relative importance of concern *i* with respect to all other concerns;
3. The priority of each concern is calculated and a relative ranking for the list of concerns is developed.

Assuming that we have four concerns called con1 to con4, the first step creates a 4×4 matrix whose entries show the relative importance of the concerns from both the stakeholders' perspective and technical value (See Table 1-a). For instance, P [1, 3] =7 would show that concern 1 is slightly more important than concern 3 and hence, we would also infer that P [3, 1] =1/7.

Table 1. Example concern prioritization steps

a) relative importance of concerns

	Con1	**Con2**	**Con3**	**Con4**
Con1	1	3	7	5
Con2	1/3	1	3	3
Con3	1/7	1/3	1	1/3
Con4	1/5	1/3	3	1

b) normalized values

	Con1	**Con2**	**Con3**	**Con4**	**Sum**
Con1	0.6	0.65	0.5	0.53	2.28
Con2	0.18	0.21	0.21	0.32	0.92
Con3	0.08	0.06	0.07	0.03	0.24
Con4	0.12	0.06	0.21	0.1	0.49

Once the matrix is created, AHP is executed to estimate the eigenvalues of the matrix (Table 1-b). Since we have four concerns, the division of the last column (Sum) will result in the relative significance of each concern: [0.58, 0.23, 0.06, 0.13]. This shows that the ordering between the four concerns is con1>con2>con4>con3. The stakeholders will have the chance to filter out the lesser important concerns based on the ranking outcome.

3.3.2 Feature Ranking Stage. This steps aims at ranking the relative importance of the qualifier tags of the remaining concerns and uses them to rank the available features within the feature model. This will provide a final ranking over the most important concerns and their most significant qualifier tags. Given such a ranking, each feature can be prioritized based on the significance of its annotated concerns. So, the idea is that features that have more important concerns attached to them are more important than the others.

The process for ranking qualifier tags is similar to ranking concerns where qualifier tags sit on the rows and columns of the developed matrix, instead of concerns.

1. Let *V′ = {t | t is possible values of c and c ∈ C′}*;

2. A Matrix $M_{|V'|\times|V'|}$= *{M [i, j] =β | 1 ≤ i, j ≤ |V'| and β is relative importance of the tag concern i to tag concern j}* is developed;
3. AHP is executed over the matrix M where $M_{i,*}$ shows the relative importance of qualifier tag i with respect to all other tag concerns;
4. Rankings of features are calculated by applying a predefined function (i.e. minimum, maximum, or mean) on the qualifier tag value ranks for each feature. The predefined function is a function that is used to select a rank for a feature when a feature has more than one concern. This function is defined by the domain engineer for the features based on stakeholders' input.

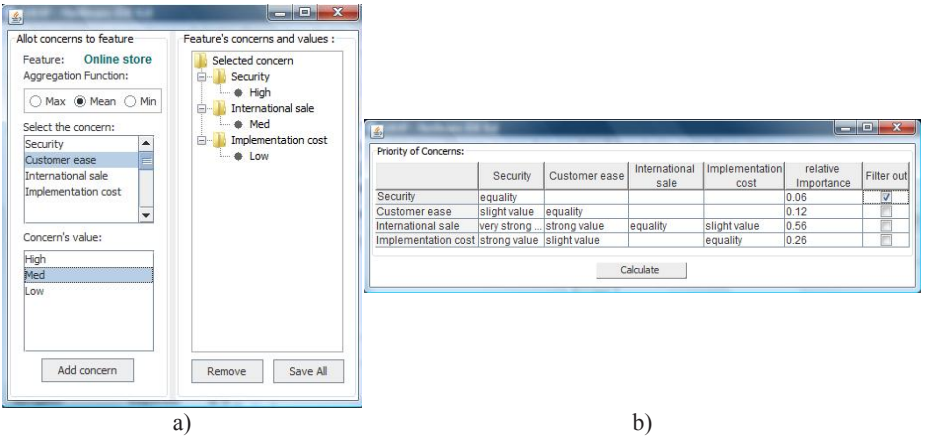
One important point to note is that some features might be annotated with more than one concern, e.g., a feature can be described as having a high performance (with the relative importance of 0.54) and low security (with the relative importance of 0.1). This annotation means that the feature will provide high system performance; a concern liked by the stakeholders (0.54), but at the same time may cause security problems, which is not liked by the stakeholders (0.1). The decision on which value to assign to the feature would depend on the modelers' strategy. A conservative modeler would fear for a security breach as a result of the inclusion of this feature and would hence take a minimization strategy and would select the lower value as the representative importance for this feature (0.1), but a modeler designing the application for an already secure environment, might adopt a more avant-garde strategy and assign higher importance (0.54) to the feature. It is also possible to take the average value as the representative (0.32).

3.3.3 Feature Model Specialization Stage. After ranking the available features based on their relative significance to the target application stakeholders, we start conducting feature specialization steps iteratively. Each specialization step uses feature ranks to select the most appropriate features for that specific application. During the different iterations of the feature model specialization stage, selection or filtration of features can be performed based on the developed rankings in the previous stage (i.e., the feature ranking stage). For this stage, we adopt the six steps introduced in [6] to limit the configuration space and adapt some of the steps of the adopted process. The steps defined in [6] are as follow: i) *Refining feature cardinalities***:** it eliminates or decreases the cardinality of features; ii) *Refining group cardinalities***:** it decreases the interval of possible grouped features that can be chosen within a group; iii) *Removing a grouped feature from a feature group:* this step removes one of a group sub-features with all of its decedents. In our process, this step removes the features which have the least importance calculated by the feature ranking stage. iv) *Assigning an attribute value:* values are assigned to uninitialized attributes during this step; v) *Cloning a solitary sub-feature*: this step provides the possibility of cloning a feature as well as its entire sub-tree.

Feature model specialization is performed iteratively. It gradually moves the specialized feature model towards its final configuration. The main contribution of our S-AHP process is that it provides suitable rankings for the available features that can be selected by the stakeholders in each stage. This ranking of features is based on the business objectives and high-level goals of the stakeholders and therefore facilitates the feature selection process.

Finally, in order to support the proposed configuration process, we have created a prototype extending the Feature Model Plug-in (*fmp)* [13] with required functionality

for managing the concerns and supporting our configuration process. *fmp* is a widely used plug-in for feature modeling and configuration. A screenshot of the tool's dialog for editing concerns and their qualifier tags is given in Figure 2a, while the part of the *fmp* extension for filtering out the most important concerns is shown in Figure 2b. Similarly, the tool allows users to effectively perform the other steps of S-AHP.



a) b)

Fig. 2. S-AHP extension of the Feature Plug-in: a) Editing concerns and associating with features; b) Concerns prioritization

4 Process Changes in Software Product Line Methodology

Software product line methodologies have commonly defined two lifecycles, namely *Domain Engineering* and *Application Engineering* [10]. In domain engineering, the common assets, family reference architecture and the variability models are developed. Afterwards, in the application engineering lifecycle, the common assets are reused and variability models are configured to produce a product family. By applying our approach for modeling the concerns within the feature models, some slight changes need to be made on the processes within the software product line lifecycles. In order to provide a clear understanding of these changes, we mention the phases in each lifecycle and highlight our changes and extensions in each of these phases. Due to the limited space for this paper, Table 2 only summarizes the activities added by S-AHP to the well-known software product line lifecycles.

5 Case Study

The online store is a commonly used example in the literature. This example is employed to represent the behavior of feature modeling frameworks [12]. Figure 3 depicts a small feature model designed to depict some of the aspects of such an online store. In this model, the features have been annotated with four concerns, namely security, customer ease, international sale, and implementation cost. The concerns are

qualified using high, medium and low values; therefore, the modelers are able to express their belief with regards to the features in the context of these concerns. The concerns and their qualifier tags are shown in the lower part of the leaf features and the legend explaining their interpretation is placed in the top left part of Figure 3. For instance, using cash as a payment method has been considered to be highly secure, terrible for international sales and inexpensive from an implementation perspective.

Table 2. Process changes in software product line process by S-AHP

Life-cycle	Phase	Traditional Activities	Added activities by S-AHP
Domain Engineering	Product line scoping	• Product portfolio scoping • Domain scoping • Asset scoping	• Examine strategic managements' viewpoints • Identify major concerns
	Domain Requirements Engineering	• Family requirements elicitation • Family requirements analysis • Family requirements specification, • Family requirements validation and verification	• Refine concerns (break down to sub-concerns) • Define concerns tag values
	Variability Modeling (Feature Modeling)	• Identifying features • Identify feature relation • Model features	• Annotate features with concerns • Assign tag values to features concerns, if it is determined. • Define aggregation function when there are more than one concern in each feature
	Domain Design	• Design reference architecture • Detail design of assets	• Update feature concerns and their values based on design information
	Domain Realization and Testing	• Make/buy/mine/commission • Test	• Update feature concerns and their values based on implementation information
Application Engineering	Application Requirement Engineering	• Application requirements elicitation • Application requirements analysis • Application requirements specification • Application requirements validation and verification • Reuse family requirement model	• Indicate relative importance of concerns for the specific product
	Application Design	• Binding variability based on application requirements • Verify and Validate specialized feature model • Automatically create application	• Ranking the concerns • Ranking tag values • Ranking features • Bind variables based on the provided ranks
	Application realization and testing	• Complete application • Test Application • Deploy application	N/A

Furthermore, we can assume that the feature model is accompanied by the following four dependency constraints: 1) cash payment implies pickup shipping; 2) credit card payment implies automatic fraud detection; 3) never changing the password implies the inclusion of special characters in the password; and 4) manual fraud detection excludes credit card payment. These constraints will be automatically applied if any of their antecedents are included in the feature model through the specialization process. Now, supposing that an actual online store needs to be developed based on the requirements, needs and goals of its target audience, S-AHP can be used to select the best set of features. In case AHP is used, since there are 20 open features in the model, 190 comparisons need to be made. As will be shown S-AHP requires a much less number of comparisons. It should be noted that in a real-world large-scale feature

model, the number of features and concerns are much higher. In the first step, the concerns are ranked through a pair-wise comparison process as shown in Table 3.

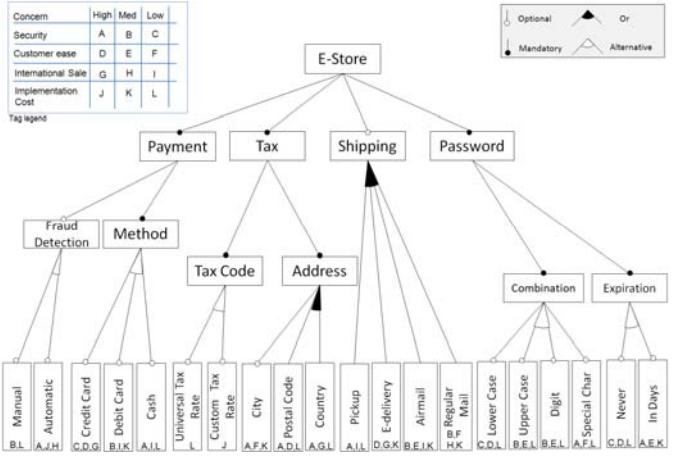


Fig. 3. A small feature model for an online store

We can consequently calculate the relative importance of the concerns with each other, which shows the following order (Table 3): International sale>Implementation cost>customer ease>security. It can be inferred that the stakeholders are interested in focusing on the international sale aspect and implementation costs of the product while customizing the feature model. It is possible to filter the less important concerns and proceed to the second stage. Based on the filtration, the new matrix (Table 4) is developed by inquiring the stakeholders about their preferences on the selected concern's qualifier tags (see Figure 3 for letter interpretations).

Table 3. Relative importance of concerns as well as the final concerns ranks

	Security	Customer ease	International sale	Implementation cost	Normalized Sum	Importance
Security	1	1/3	1/7	1/5	0.22	0.06
Customer ease	3	1	1/5	1/3	0.4802	0.12
International sale	7	5	1	3	2.2305	0.56
Implementation cost	5	3	1/3	1	1.0505	0.26
Sum	16	9.33	1.67	4.5	3.9812	1

It is clear from the developed ranking that the stakeholders are interested in having high international sale in their online store. In addition, they are not interested in spending a lot of money on technical implementation issue; therefore, they should select those features which would result in high international sale with low implementation cost. Based on this information, a ranking of features can be simply developed. As an example and based on averaging the qualifier tag values of the concerns attached to each feature, the manual fraud detection method (0.3) would be more desirable than its automatic (0.1) counterpart, and the credit card-based payment method (0.4) is also more attractive than the other methods (cash: 0.15; debit: 0.05). So, the modelers would select manual fraud detection and credit card based payment features;

however, this selection would be in conflict with one of the dependency rules (manual fraud detection excludes credit card payment). In this case, the modelers will probably decide to remove the manual fraud detection feature since it is less important than the credit card-based payment method (0.4>0.1) and also include automatic fraud detection to satisfy the dependency rule (credit card payment implies automatic fraud detection). This decision can conclude the first stage of the configuration process, which results in the selection of the appropriate features for payment method and fraud detection; hence, peer features to the credit card payment method and the automatic fraud detection that have not been selected in this stage will be removed from the online store feature model.

Table 4. Tag values ranks calculated by S-AHP

	G	H	I	J	K	L	Importance
G	1	5	7	7	5	3	**0.4**
H	1/5	1	5	5	3	1	**0.15**
I	1/7	1/5	1	3	3	1/7	**0.01**
J	1/7	1/5	1/3	1	1/5	1/7	**0.04**
K	1/5	1/3	1/3	5	1	1/5	**0.1**
L	1/3	1	7	7	5	1	**0.3**

The first specialization in the staged configuration of the online store feature model can be seen in Figure 4. The feature model can be further specialized through more rounds of refinement and feature selection based on the developed ranking, so that the most desirable configuration of the feature model is achieved. An important point here is the significant difference between the required number of comparisons in AHP and S-AHP. For this rather small feature model, AHP would require 190 comparisons, while S-AHP needs 6 comparisons in its first layer and 15 comparisons in the second layer, making it 21 comparisons in total. It is clear that as the size of the feature model grows larger, the efficiency and utility of S-AHP will become even more noticeable.

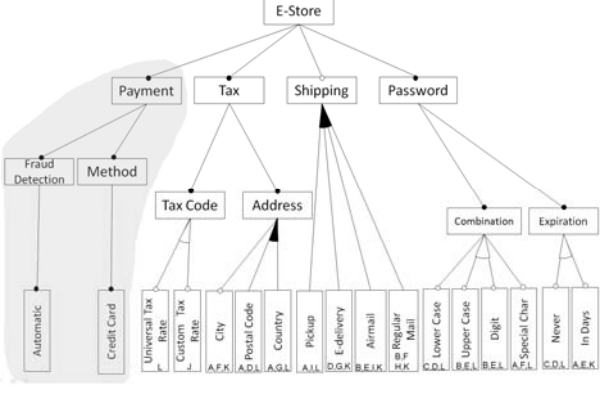


Fig. 4. A specialization of the online store feature model. The shaded area shows the result of the specialization performed in the first stage.

6 User Evaluation

It is also important to see how S-AHP is perceived by software designers and practitioners to be applied to real world problems. For this purpose, we have evaluated the usefulness of S-AHP by providing S-AHP to software practitioners and asking for their feedback on how helpful they find it for the tasks that they usually perform. Eleven software practitioners were invited to participate in the study whose range of software development experience was from 6 to 15 years, with an average of 9.5 years. The participants expressed that they were familiar with various application design and planning methodologies including SSADM, OOP, and RUP, and were involved in major industrial design projects spanning the areas of geographical information systems, accounting, mobile applications, customer relation management and others. The participants were asked to provide their evaluation of the S-AHP method from their perspective for its usefulness, efficiency, easiness for use, and practicality. The participants were asked eight questions by using a seven-level Likert scale where 1 being strongly unfavorable to the concept and 7 being strongly favorable to the concept. The following eight questions were asked. We provide the average value that S-AHP received for each of these questions along with the question: 1) It is simple to use S-AHP: ***6.0***; 2) I can effectively complete my work using S-AHP: ***5.0***; 3) I am able to complete my work quickly using this method: ***5.1*** ; 4) I am able to efficiently complete my work using this method: ***5.3***; 5) It was easy to learn to use S-AHP: ***6.1***; 6) I believe I became productive quickly using S-AHP: ***4.8***; 7) I believe S-AHP increase the chances of selecting the most suitable/important software features: ***5.8***; 8) Overall, I am satisfied with this method: ***5.8.*** The participating software practitioners in the study felt that S-AHP is effective in helping them more conveniently choose the best set of software features, and that its simplicity makes it practical for real-world software development environments. In addition, it was pointed out that the proper selection of the most relevant set of concerns for each project has effect on the final set of selected features; therefore, close attention needs to be made in order to select the best concerns.

7 Related Works

The research community has put much emphasis on developing methods for the syntactical validity checking of model configurations. These methods mainly focus on forming grammatical correspondences for the graphical representation of feature models and perform automated syntactical analysis (e.g. using AI configuration techniques such as SAT or CSP solvers) based on the model dependency rules and constraints [12][14]. However, considering the strategic objectives of the stakeholders and the specific domain requirements can create a semantic validation process that will ensure that the requisites of the target audience is met [15]. Such semantic validation process can complement syntactical consistency checking methods in order to create a valid and at the same time useful final configuration of a feature model.

Some researchers have proposed to annotate features of a model with priority information, which will be later used to select high priority features at specialization time [2]. The idea of prioritization of features is interesting, but it does not seem to be

suitable for the context of feature models and product families. This is because feature priorities change based on the context where the feature model is being specialized and configured; therefore, priority values for features should be developed during specialization, which is supported by the S-AHP method. As an example, a password control feature for an application would take high priority on a network installation, while it would have less priority in a standalone unique installation. Therefore, single priority values are not sufficient and methods that support dynamic priority assignment are required.

Closely related to the idea of feature selection, the requirement engineering community has significantly contributed to the development of requirement prioritization techniques for requirement selection [16][17]. In contrast to feature modeling techniques, requirement specifications are commonly defined in an unstructured (free-text) or semi-structured forms; therefore, semantic validation of requirements by a human requirement engineer is more viable than automatic syntactic correctness checking. In this context, one suggestion for prioritizing requirements has been the use of requirement priority groups as a way to nest similar requirements together and create an internal rank for requirements in each group. This approach is not so suitable for feature models, since the structure of the feature model tree is of high importance while the decision about the priority and specialization of the feature model is being made; therefore, creating priority groups that would become feature tree structure oblivious might result in irrelevant feature model configurations.

Similarly, hierarchical Analytic Hierarchy Process (AHP) [8] has been proposed in the requirement engineering literature to create a structure for interrelated requirements. In this method, only requirements at the same layer are compared with each other and hence reduce the total number of required comparisons in contrast to AHP; however, this method is not so suitable for feature models since the features in the interim layers of the feature tree are usually not included in the feature comparison process, which removes the need for a hierarchical structure; therefore, this method does not actually reduce the number of required comparisons for a feature model. From a different perspective, the Bubble sort technique has been used order the requirement statements. Bubble sort is in essence very similar to AHP with the slight difference that requirement comparisons are made to determine which requirement has a higher priority, but not to what extent. It is clear that Bubble sort suffers from similar issues to AHP, e.g., the large number of required comparisons. There have been proposals to reduce the number of required comparisons in comparison-based techniques, which are generally referred to as incomplete pair-wise comparison methods [9]. These techniques are based on some local and/or global stopping rule, which determines when further comparison will not reveal more useful information with regards to the prioritization of the options. Such techniques can be beneficial if used along with techniques such as AHP, S-AHP, Bubble sort and others. Additionally, (hierarchical) cumulative voting has been used to prioritize requirements where top vote-getter requirements are prioritized higher than the others. One of the drawbacks of that approach is that as the number of requirements (options) increases, it becomes very hard for the stakeholders (voters) to select the best voting tactic, which would reveal their preferences about the highest priority requirements [18].

Other techniques such as our own experience with the formulation of multi-attribute utility theory for architectural tradeoff decision making have shown that

these methods are too complicated and hard to understand by practitioners [15,19]. This observation necessitates the development of an easy–to-use and straightforward yet efficient method for feature selection.

8 Concluding Remarks

Proper feature selection and feature model customization requires the systematic consideration of the stakeholders' needs and objectives as well as the constraints and dependency rules of the feature model. To this end, we have introduced a useful set of methodical activities to support feature selection, called S-AHP. The aim of S-AHP is to communicate with the stakeholders and understand their priorities with regards to business objectives and high-level goals and to use this information to find the most important features of a feature model for the stakeholders. Here, we outline how S-AHP satisfies the requirements of an efficient feature selection method based on the challenges and characteristics introduced earlier in the article:

1. S-AHP is able to tame the relatively large number of comparisons needed for developing the ranking between the features. This is achieved by a layered pair-wise comparison of the feature annotations, namely the concerns and their qualifiers. We showed that for a relatively small example a reduction from 190 to 21 comparisons was reached.
2. It overcomes feature incompatibility and the comparison problem by introducing the concept of concerns. Since features are annotated with multiple concerns they can be indirectly compared and ranked. This is due to the fact that concerns are comparable, and their priorities can be effectively propagated to their corresponding features.
3. This method prevents the appearance of different comparison scales by clearly employing and defining default comparison scales (1, 3, 5, 7, 9). A strategy to use these values can serve as a consistent comparison scale to be employed by the stakeholders and the practitioners.
4. The activities within S-AHP are very easy to perform and are based on a simple pair-wise comparison method. The relative simplicity of this approach is advantageous because it does not add complexity to the complicated feature selection process. Furthermore, it can be quickly and inexpensively implemented in a spreadsheet program such as Microsoft Excel without the need to purchase commercial decision support tools.
5. S-AHP is also able to effectively communicate with the stakeholders at a higher level. This is achieved thanks to the concerns and their qualifier tags. With the employment of concerns, the stakeholders do not need to be aware of the structure or meaning of the feature models, but would be able to understand whether the customized feature model satisfies their requirements and objectives.

In summary, S-AHP is a simple yet effective approach for ranking and filtering various features of a product family, which expedites the product specialization and staged configuration of a feature model. Its main advantages are its simplicity, practicality and its involvement with the target audience and product stakeholders that allows for the alignment of the final product with the strategic objectives and goals of

the stakeholders. Furthermore, due its simplicity the development of tool support for this approach is quite easy and very inexpensive, and even many freely available tools for AHP can be easily used in the context of S-AHP. Currently, S-AHP only supports for a uni-dimensional pair-wise comparison of the concerns where the comparison of two concerns is represented by a single value from taken from the comparison scale. As future work, we are interested in investigating whether multi-dimensional extensions of S-AHP would be beneficial that would simultaneously incorporate matters such as cost and value in the comparisons in order to perform tradeoff decision making.

References

[1] Clements, P., Northrop, L.: Software product lines: practices and patterns. Addison-Wesley Longman Publishing, Amsterdam (2001)
[2] Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Soft. Proc. Improv. and Practice 10, 7–29 (2005)
[3] Heymans, P., Schobbens, P., Trigaux, J., Bontemps, Y., Matulevicius, R., Classen, A.: Evaluating formal properties of feature diagram languages. IET Soft. 2(3), 281 (2008)
[4] White, J., Dougherty, B., Schmidt, D.C., Benavides, D.: Automated Reasoning for Multi-step Software Product-line Configuration Problems. In: SPLC 2009 (2009)
[5] Boskovic, M., Bagheri, E., Gasevic, D., Mohabbati, B., Kavinai, N., Hatala, M.: Automated Staged Configuration with Semantic Web Technologies. International Journal of Software Engineering and Knowledge Engineering (in press)
[6] Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration Through Specialization and Multi-Level, Dep. of Electrical and Computer Eng., University of Waterloo (2004)
[7] Sommerville, I., Sawyer, P.: Viewpoints: principles, problems and a practical approach to requirements engineering. Annals of Software Engineering 3, 101–130 (1997)
[8] Saaty, T.L.: The Analytic Hierarchy Process. McGraw-Hill, New York (1980)
[9] Karlsson, J., Olsson, S., Ryan, K.: Improving Practical Support for Large-scale Requirement Prioritising. Requerments Engineering 2 (1997)
[10] Linden, F.J., Schmid, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer, Heidelberg (2007)
[11] Linden, F., Phol, K., Bockle, G., Sikore, E., Gunter, B.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
[12] Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, p. 7. Springer, Heidelberg (2005)
[13] Czarnecki, K., Kim, C.H.: Cardinality-based feature modeling and constraints: A progress report. In: International Workshop on Software Factories (2005)
[14] Batory, D., Benavides, D., Ruiz-Cortes, A.: Automated analysis of feature models: Challenges ahead. Communications of the ACM 49, 47 (2006)
[15] Bagheri, E., Ghorbani, A.A.: The analysis and management of non-canonical requirement specifications through a belief integration game. Knowledge and Information Systems 22, 27–64 (2009)
[16] Perini, A., Ricca, F., Susi, A.: Tool-supported requirements prioritization: Comparing the AHP and CBRank methods. Inform. and Soft. Tech. 51, 1021–1032 (2009)

[17] Aurum, A., Wohlin, C.: Eng. and Managing Software Requirements. Springer, Heidelberg (2005)
[18] Berander, P., Jönsson, P.: Hierarchical Cumulative Voting (HCV) – Prioritization of Requirements in Hierarchies. Int'l. J. Soft. Engi. & Know. Eng. 16, 819–849 (2006)
[19] Makki, M., Bagheri, E., Ghorbani, A.A.: Automating Architecture Trade-off Decision Making through a Complex Multi-attribute Decision Process. In: Morrison, R., Balasubramaniam, D., Falkner, K. (eds.) ECSA 2008. LNCS, vol. 5292, pp. 264–272. Springer, Heidelberg (2008)

Streamlining Domain Analysis for Digital Games Product Lines

Andre W.B. Furtado, Andre L.M. Santos, and Geber L. Ramalho

Centro de Informática (CIn) – Universidade Federal de Pernambuco (UFPE)
Av. Professor Luís Freire, s/n, Cidade Universitária, CEP 50740-540, Recife/PE/Brazil
{awbf,alms,glr}@cin.ufpe.br

Abstract. Digital games and their development process are quite peculiar when compared to other software in general. However, current domain engineering processes do not addresses such peculiarities and, not surprisingly, successful cases of software product lines (SPLs) for digital games cannot be found in the literature nor the industry. With such a motivation, this paper focuses on streamlining and enriching the Domain Analysis process for SPLs targeted at digital games. Guidelines are provided for making Domain Analysis tasks aware of digital games peculiarities, in order to tackle the challenges of and benefit from the unique characteristics of such a macro-domain. A case study for an SPL aimed at arcade-based games is also presented to illustrate and evaluate the proposed guidelines.

Keywords: digital games development, software product lines, domain analysis.

1 Introduction

The idea of family-based production strategies was first introduced by David Parnas in 1976 [1]. According to him, a set of programs is considered to constitute a family whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members. Building on top of that, Czarnecki & Eisenecker [2] advocated that the first step in the transition from single systems to system families is to adopt a domain engineering or software product line (SPL) process.

Many cases of successful software product lines in practice can be found for different domains [3], such as consumer electronics, printing machines and avionics. However, this is not true for domains belonging to digital games development, a field typically characterized by ad hoc, low-level development [4]. In fact, current SPL and domain engineering processes do not address the peculiarities and specific constraints related to digital games development. This is concerning since such an industry, in a consistent manner, presents expressive numbers that consolidate its relevance. For instance, both computer and console games were responsible in 2008 for 11.7 billion dollars in sales [5], while the video game industry (at least in the U.S.) has a 19% growth year over year.

As a starting point to conciliate Domain Engineering processes with digital games development, this paper presents a set of guidelines for streamlining multiple Domain

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 316–330, 2010.

Analysis tasks towards empowered digital games SPLs. Our focus is not only on establishing a clear understanding on what game development peculiarities should be tackled as Domain Analysis challenges, but also on identifying where Domain Analysis can benefit from the unique characteristics of such a macro-domain.

The remainder of this paper is organized as follows. Section 2 elaborates on the peculiarities of digital games development. Section 3 revisits traditional Domain Analysis tasks in the light of digital games development, providing guidelines as appropriate. Section 4 presents a case study of the proposed guidelines for an arcade-based games SPL. Finally, Section 5 presents conclusions about the research and points out future directions.

2 The Peculiarities of Digital Games Development

Neward [6] points out that there are a number of challenges coming up that we cannot solve with our current set of languages and tools, and that we stand on the threshold of a "renaissance in programming languages". Greenfield et al. [7], for instance, advocates that the total global demand for software is estimated to grow by an order of magnitude over the next decade: *"design patterns and specialized tools demonstrated limited but effective knowledge reuse; however, without deeper increases in productivity, total software development capacity seems destined to fall far short of total demand"*.

Nonetheless, the expectations on digital games are already extremely high [8]. Innovative hardware, advanced business models, applicability to multiple domains (entertainment, education, training, etc.) and innovative gameplay make digital games to be perceived as one of major streams where bleeding-edge technologies and ideas are showcased. This way, it seems that the exponential growth of the total global demand for software is a trap waiting for the game development industry, since:

- The hardest part of making a game has always been the engineering [4];
- Many game developers struggle with component integration and managing the complexity of their architectures, while expanding deadlines and escalating costs have notoriously plagued the game industry [8].
- Game development is a field typically characterized by ad hoc, low-level development [9]. Historically, excessive high performance constraints forced digital games development to trade more refined software engineering techniques for a result-oriented but less organized development process, as well as reusability for in-house development, in a methodology that became known as "pedal to the metal" [10].

The many ways in which the digital games domain differ from other software in general are also an indication that domain engineering processes, if targeted at digital games development, should be streamlined in order to become more effective. Challenges and opportunities include:

- The concept of "genres" is extremely popular in the digital games macro-domain. It is commonly used as an attempt to define taxonomies into whose branches games can be classified. However, genres are blurry and imprecise:

there is no agreement on a universal set of genres neither on the individual meanings of specific genres.

- The development of a digital game is not a direct outcome of user requirements or business needs, which may not even exist. Games are not focused on solving user problems, but to entertain them. On the other hand, psychochemical requirements such as immersion, surprise, delight and nostalgia are present in digital games. This way, traditional Requirements Engineering cannot be applied as is to game development. For example, the well-known concept of "use cases", with well-defined roles, flows and input/output artifacts may not make sense to a game development process. Game Design documents and experimentation processes are more realistic in this area.
- User interaction is unique in digital games, especially when compared to other types of software generally based on mouse, keyboard and limited graphical interface standards (e.g., windows-based GUIs). In digital games, adherence to standards is many times trumped by the desire to provide innovative experiences.
- An impressively huge diversity of game instances, even from decades ago, is still in use today in the digital games domain. In contrary to the majority of other software domains, nostalgia causes "retro" instances to be kept alive for generations. Many samples are also available due to other reasons peculiar to game development, such as the diversity of platforms, prototyping culture and user-generated content. This translates into rich, valuable and available input for designing future software in the domain.
- Studies on the applicability of software product lines in the game development domain are still incipient. Industrial secrecy to support competitive advantage is very high in the domain, since such projects involve great investments. For example, it is difficult to find comprehensive studies about the applicability of design patterns in game engines [11].

3 Addressing Game Development Peculiarities via Domain Analysis

In the context of a Domain Engineering process, many of the digital games development peculiarities presented in the previous section can, and should, be addressed during the Domain Analysis phase. Such a phase was first introduced by Neighbors [12] as *"the activity of identifying the objects and operations of a class of similar systems in a particular problem domain."* It can also be defined [13] as a process by which information used in developing software systems is identified, captured and organized with the purpose of making it reusable when creating new systems. This section discusses and supplements some vital Domain Analysis tasks in the light of digital games development.

3.1 Envisioning and Scoping the Game Domain

The great diversity of games created so far has turned the digital games universe into an extensively broad domain. Therefore, creating a SPL targeted at digital games in

general, ranging from 2D platform games to 3D flight simulators, constitutes a too broad and ineffective endeavor. In such a scenario, the SPL processes, its tools and assets would not be able to fully exploit SPL benefits such as component reuse and assemblage, or domain-specific languages (DSLs) expressiveness. In other words, a narrower subset of games should be chosen.

One of the most often used attempts to narrow down and classify digital games are game genres, which together compose a game taxonomy. Examples of popular genres are adventure, role-playing (RPG), shooter, simulation and strategy. Nevertheless, defining genres can be a quite difficult task as many people have different opinions on the meaning of a genre or various ways of stereotyping them [14]. Likewise, it is not rare for a game to fall into more than one category. Some authors even argue that describing different types of games requires different dimensions of distinctions (narratology, ludology, simulation, gambling, etc.), i.e., orthogonal taxonomies which allow design concerns to be separated.

In fact, classifying games into genres is a difficult task not only because a game can be hybrid, but also due to the fact that some genres are "horizontal", such as for casual games, educative games, serious games, adult games and advergames. The high evolution speed experienced by game genres is also an issue. Crawford [15] points out that, due to the dynamic nature of game taxonomies, they can be expected to become obsolete or inadequate in a short time.

Moreover, there is a lack of consensus at the level of the classification schemas, some being more popular than others. For instance, some schemas are largely semiotic, while others rely more strongly on configurative patterns of interface and mechanics. In short, due to a general lack of commonly agreed-upon genres or criteria for the definition of genres, classification of games are not always consistent or systematic and sometimes outright arbitrary between sources.

One interesting example of game genre disagreement is the *"Shoot'em up"* genre, where the player controls a single character, often a spacecraft, shooting large numbers of enemies while dodging their attacks. Such a genre encompasses various types or subgenres and critics differ on exactly what design elements constitute a shoot 'em up game. Some restrict the definition to games featuring spacecraft and certain types of character movement; others allow a broader definition including characters on foot and a variety of perspectives.

As a consequence, game genres alone are not reliable enough to describe and envision a SPL domain aimed at digital games. Solely relying on game genres is a trap that very likely will lead to ambiguities and difficulties during Domain Engineering activities, such as when selecting domain samples to analyze during Domain Analysis. For example, a loosely defined "Racing Games SPL" can mean different things: third-person racing games, arcade racers or racing simulators, not to speak about other possibilities such as street racing games, off-road racing games and others specifically focused on motorcycles, trucks, speedboats or even horses.

Instead of embracing the challenge of reaching the perfect game genre taxonomy, we are actually focused on ensuring that digital games development can benefit from systematic planning and management in the context of a specific application family. In such an ambiguous universe of game genres, we propose an alternative solution, which has the added benefit of bootstrapping the domain scoping process: the definition of a **game domain vision** to formerly describe a game family that share functionality. Such

a vision intends to reach a common agreement on what ultimately defines the domain to be approached, independent of game genres. It does so by means of the following tasks, detailed in the next sub-sections: setting expectations for core game dimensions, establishing a negative scope, identifying target platforms and the creating a vision statement.

Setting expectations for core game dimensions. If from one hand the digital games macro-domain is broad, requiring game domain analysts to narrow it down so that it becomes viable to the context of a software product line, from the other hand such a macro-domain is still more specific than the broader "software" concept, lying somewhere in between. Therefore, setting expectations for core game dimensions which are ubiquitous to digital games development, yet not too specific neither too generic, can be used to drive the envisioning phase of a digital games SPL.

The list below presents suggested digital games core dimensions to be explored in the creation of digital games SPLs:

1. **Player:** concepts related to the game player(s), such as number of players, co-playing modes (e.g., in turns or simultaneously, cooperative or "deathmatch", etc.), score, high-score, lives and others. This should not be confused with "main character" entities controlled by the players.
2. **Graphics:** what players are supposed to see, including the world view (2D, isometric, first-person, etc.), heads-up displays (HUDs) and eventually more advanced techniques such as particle systems to simulate fire, dust, rain, etc.
3. **Flow:** how the domain games evolve as perceived by players, encompassing levels, phases, missions, screens, rooms, scenes, etc.
4. **Entities:** the underlying types and mechanics of beings and things that players are supposed to control and interact with.
5. **Events:** triggers and reactions that drive the behavior of the world and entities of the domain games.
6. **Input:** how players provide input to interact with the domain games, encompassing (a combination of) devices and eventually more advanced options such as speech recognition and controller-free systems.
7. **Audio:** what audio feedback players are supposed to get from the domain games, including sound effects, background music and optionally more advanced technologies such as 3D sound, speech synthesis (text-to-speech) and special effects (echo, pitch, reverb, bullhorn, etc.).
8. **Physics:** the physical mechanics of the produced games, including collision detection and acting forces.
9. **Artificial Intelligence:** artificial intelligence behaviors performed by entities and the world of the domain games.
10. **Networking:** whether the domain games are standalone applications or can interact with servers (to store high scores, for instance) and/or other running game instances.
11. Any other **custom core game dimension** that applies to the domain. For instance, an important core game dimension that constrains role-playing games (RPGs) is the *Battle System*, which determines whether fights against enemies happen in turns or as real-time action, randomly or planned, etc. On the other hand, card games can have special constraints

based on card decks, such as the number of decks, usage of full or partial decks, etc.

The expectations for such core game dimensions come from multiple (and many times informal) sources, such as expertise from domain experts, previous knowledge from domain analysts, trends and influences from successful game titles, requirements from game developers or designers and an overall assessment of the high-level goals of the game SPL. By no means should the resulting set of expectations be considered final or totally accurate. On the contrary, it will very likely be modified and refined by subsequent activities and iterations.

Establishing a negative scope. A negative scope is focused on stating expectations that will not belong to the specific game domain being defined, and therefore will have no built-in support from the SPL assets, such as DSLs and transformations. This task is especially useful regarding expectations that the game SPL customers (game developers and designers) would implicitly take for granted, but are out of scope by design. For example, a SPL focused on a racing games domain may explicitly state, through its negative scope, that refueling and campaign modes are out of the SPL expectations.

Initially, negative expectations can be derived from core game dimensions. Later, the Domain Analysis activities can refine the negative scope, explicitly stating the domain features that should not be taken into account. The negative scope, however, does not avoid SPL customers to add missing features to their games as extensions and customizations to the SPL. In fact, the SPL can still provide extensibility mechanisms ("hooks"), such as parameterization, partial classes, events, sub-classing, polymorphism, dynamic class loading, dependency injection, etc. [16]. For example, a game entity movement can be originally restricted to a built-in set or formula (8-directions via the arrow keys, mouse position-based, etc.), but be extensible enough to enable game developers to define alternative possibilities (such as bouncing) by overwriting methods that move entities around.

Identifying target platforms. We suggest the game domain documentation to include the target platforms to be supported by the SPL, such as consoles, mobile devices, PC (running under a client operating system), web (running under a browser), digital TV, etc. Constraints on the target platforms, such as the need for a specific browser technology (Flash, Silverlight, etc.), operating system or runtime can also be described if the information is available or as a result of refining the domain envisioning in future iterations. Product portfolios [17] can be used to describe all families supported by the game SPL, by means of their target platforms. A product map can also be conceived in order to map capabilities and restrictions of core game dimensions into the target platforms. For example, one of the target platforms in a mobile games domain may have its graphics expectations restricted to some specific screen sizes.

Creating a vision statement. A comprehensive yet concise vision statement summarizes the essence of a game domain vision. Although the vision statement is established only after other game domain vision elements, it will generally be presented first when introducing the game SPL to stakeholders, as it happens with an executive summary of a business plan.

Additional guidance is provided to envision and scope a digital games SPL:

- As highlighted by agile development methodologies, analogies are a powerful mechanism to facilitate understanding.
- The extent of the vision is key to determine the success of a digital games SPL. If it is too broad, it may attempt to encompass too many genres, therefore it will have excessive variability resulting in ineffective game SPLs. On the other hand, if the vision is too narrow, it may be difficult to reuse processes, components and tools, making it harder to achieve a return on the game SPL upfront investments.
- The vision should not depend on game-world content, unlike other works of fiction such as films or books. For example, an action game is still an action game, regardless of whether it takes place in a fantasy world or outer space [18].
- The vision should be comprehensive, but not meticulously precise. False positive samples (games that may fit into the vision but are later discarded by Domain Analysis) can still exist. As the vision gets refined by Domain Analysis, the domain boundaries are made clearer.
- The domain envisioning elements proposed by this section (expectations for core game dimensions, negative scope, etc.) can be customized by the game SPL in order to more properly fit the contents and layout of Game Design Documents already in use by the organization. In fact, such envisioning elements can be seen as a simplified version of a meta-Game Design Document, to be refined by game SPLs and instantiated during application development.
- If game domain analysts foresee that different player profiles (or even other types of end-users) will be addressed by the game SPL, Personas [19] can be modeled and become part of the vision. Examples of Personas include teachers and students in educative games, hardcore and casual gamers for games with multiple levels of difficulties, different content generator roles for customizable games, etc. The specific needs of each Persona can be used as input when conceiving game SPL assets later on.
- Game domain analysts have an additional dilemma to deal with when envisioning the domain. If from one side each wave of games is attempting several technical feats (and experiences) that are mysterious and unproven to surprise players [4], on the other hand there should be a threshold on any core game dimension innovation, in order to avoid it from becoming a rupture. However, this should not rule out a completely new gameplay experience, especially if it is solidly based on user experience research.

3.2 Defining and Refining Game Domain Features

SPLs and software factories commonly approach the modeling of domain features by making a clear separation between the problem domain and the solution domain [7]. Modeling the former is an input to modeling the latter, which is then ultimately used to bridge the Domain Analysis to subsequent Domain Engineering phases (Domain Design and Implementation).

In digital games, however, the problem domain is peculiar since the concept of user requirement is not clear. Raph Koster, in his acclaimed book A Theory of Fun for Game Design [20], concludes that we play games because they offer "juicy patterns for our brains to consume". As the human brain is addicted to learning new patterns, it welcomes any activity that teaches it something new, until it loses the interest due to a mismatched difficult level, when it becomes too easy or too hard to assimilate new patterns. Balancing the difficulty level is already a big challenge by itself, since every human being is unique and therefore has personal skills and backgrounds that shape his/her own gameplay thresholds.

That said, the problem domain for digital games permeates the subjective topics of pattern-matching, fun, immersion, escapism, delight and competitiveness, combining areas ranging from social behavior to adrenalin/dopamine, which are better addressed by psychochemical sciences rather than Software Engineering. If in other industries the requirements and design patterns (such as "safety" and "comfort" in the automobilist industry) clearly map to the solution domain (air-bag, anti-break systems, automatic transmission, hydraulic steering, etc.), in digital games such mapping is not evident enough. Moreover, while requirements and software product lines in general evolve as a direct consequence of identifying new user needs, in digital games the evolution and innovation processes are mostly based on experimentation and creativity. This way, although it is well known that software in general is very likely to change during the development process, the churn seems to be higher for digital games as interim experimentation and exploratory results can radically change and shape the final product. For instance, in some professional game development studios, designers are not required to come up with a detailed game specification until the first playable is approved.

The implications of that in game SPLs are twofold:

- **Unless problem domain psychochemical features are refined and made more concrete, modeling them does not seem to be useful to the game SPL**, due to their subjective and cross-discipline nature. Typical problem domain feature examples for that are "appealing physics" and "nostalgia", which can only benefit from Domain Engineering processes if their understanding and underlying requirements are really made specific. However, **some non-experimental game features can still exist and be traced back from the solution domain to the problem domain**, as illustrated by Table 1 (the list can increase for domains in which games have secondary goals, such as in corporative training "serious" games). In such a case, this research suggests the same approach proposed by Greenfield et al. [7], which evolves the problem domain along with the solution domain for the SPL.
- Besides common, optional and eventually parameterizable features, the products (games) generated by a game SPL will present extended features as the result of an exploration/experimentation process. While Application Engineering processes targeted at creating instances of a given game SPL can address the common and optional features, **exploratory features need to be handled as SPL extensions. Later on, extensions can still be retrofitted to the SPL, as part of the game SPL feedback process.**

Table 1. Non-experimental features can enable tracing between problem and solution domains

Problem Domain Features	**Solution Domain Features**
Take breaks avoiding progress to be lost	Save/Load, Pause/Resume, "Continues"
Register player performance	High-scores table, achievements
Provide social interaction	Multiplayer mode (online and local)
Establish a player identity	Avatar, game elements customization
Availability (to play independent from time/space)	Mobile platforms, digital convergence (multi-device experience for a same game)
Readiness to play	Intuitive/one-click installers, zero-deployment games
Replay-value	Multiple narrative paths, multiplayer support, achievements
Low learning curve	Tutorials, scaffolding (hints and tips that stop being offered as players acquire experience)
Advertise a specific brand (typical for advergames)	Hooks for brand insertion, which can end up as patterns: background of "loading" screens, mid-action fly-outs, specific areas or canvas designated for branding, etc.
Teach or train the player on a given real-world topic	Missions and problem-solving challenges that incorporate the topic contents, notorious in serious end educative games.

"Braid" is an Xbox game that very conveniently illustrates this discussion. While it has all typical gameplay elements of a platform game, it innovates by adding time interaction and manipulation to the gameplay experience. For example, in one of Braid's phases, if the main character moves to the right, time advances for all other entities of the game. On the other hand, if the main character moves left, time goes back for other entities, i.e., they undo the actions they have previously done, such as by moving back to their original positions. Other game features are also impacted by time flow manipulation, such as the background music, which is played in reverse mode when time goes back.

While this feature unconsciously addresses many psychochemical desires of Braid players (such as the surprise element to play with time flow, along with satisfaction and re-wards from solving new sorts of time flow-based puzzles), very likely it was not conceived as the result of a well-defined Domain or Application Engineering process focused on user requirements. More probably, such a feature came from exploratory processes leveraging previous game design experience, and was gradually validated by experimentation through prototypes.

Supposing that Braid was created in the context of a platform games SPL, however, such time flow manipulation feature could be retrofitted into the SPL as part of its feedback process. The feature could be refined into more detailed and well-understood solution domain features, such as "entity actions recording", "entity actions playback" and "entity actions rollback". On the other hand, it could also impact already existing features. For example, the "background music" feature could be parameterized in order to allow the game background music to be played in reverse mode. As a practical consequence of that, SPL assets (languages, frameworks, etc.) would be adjusted to be compliant with and enact the updated feature set.

3.3 Analyzing Game Samples

A couple of peculiarities stand out when analyzing game samples. Firstly, a very important concern is to select games which are the most representative, since domain analysts do not have infinite resources. In digital games, **good indicators for game sample representativeness are re-releases ("remakes"), the number of sequels of a game and whether it received broad industry and media recognition**.

Not rarely, a game has sequels consisting of very similar titles, such as Pac-Man, Pac-Man II, Mrs. Pac-Man and derived variations. In such a case, **game domain analysts can opt for analyzing sequel titles as a single group, considering unique sequel features as extensions or variations of the original game**. If the sequel/variation games have expressive peculiarities, this may be an indication that the SPL domain can be partitioned into sub-domains.

As opposed to software in general, whose design if focused on enabling features to be easily reached and explored by end-users, many functionalities in a digital game are locked from the beginning. For instance, some levels should be cleared in order to unlock advanced levels. Such advanced levels, on the other hand, might reveal additional behavior (features) not originally noticed in previous levels. The web game RunMan: Race Around the World[1] is an interesting example: the concept of a world map, which links different playable areas, is not know by the player until all levels from the first area are cleared.

This way, it might not be trivial to explore all features of a digital game, and consequently perform Domain Analysis properly, without playing (and many times mastering) the game. The lack of specifications or access to game design documents negatively impacts such an already challenging concern. **This research suggests the following techniques for game domain analysts to overcome locked features:**

- **Enabling "god modes" or activating "cheat codes"**, if available. Those resources give enhanced powers to players such as the freedom to teleport across game levels or an unlimited number of lives, ammo, etc. However, when using such techniques, the game domain analyst should keep track of what a built-in game behavior is versus what was modified, so that the game analysis is not jeopardized. It is also worth noticing that god modes and cheat codes are, by themselves, features that can be addressed by the game SPL.
- **Exploring official and "underground" literature related to the game**. This includes strategy guides released by the publisher or others, playbooks[2], specialized magazines, reviews, online forum conversations and logs. Game wikis (including game-specific Wikipedia topics) are reasonable resources in this context, since addicted players worldwide do a very good job in documenting the behavior, scripts, characters and many other attributes of their favorite games.
- **Interviewing experienced players who master the game or the game domain**. Such players can also be interviewed for eliciting anticipated game features that do not exist yet.

[1] http://whatareyouwait.info

[2] An instruction book containing play scripts or diagramming various possible plays to be performed

While the feature model notation [21] tell variability by means of documenting whether a feature is common or optional in a domain, it cannot tell to what extent an optional feature is variable. For example, it is impossible to document, using a feature model, all possible flow configurations among the screens of the games belonging to a domain (e.g., only one screen, two screens where the first screen leads to the second, two screens configured in a loop, three screens, etc.). Since understanding how variability behaves for the domain features has a huge impact on how core domain assets such as domain-specific languages are conceived, **it is recommended that the game domain analyst annotates the feature model at least with textual information describing the variability universe for a given feature**.

3.4 Anticipating Features for a Game Domain

The identification of commonalities and variabilities should not be restricted to the features identified in the analyzed samples. Game domain experts, together with game domain analysts, may foresee innovative features that could enhance the generated games and therefore extend the feature model with new anticipated features, or even new anticipated sub-domains. Therefore, besides existing applications, both future applications (i.e., applications whose goals are rather clear, but development has not yet started) and potential applications (i.e., applications for which no clear requirements or goals exist yet, but that are seen as relevant) should be considered in the Game Domain Analysis.

Besides traditional brainstorming and brainwriting sessions involving the game SPL stakeholders (domain experts, players, etc.), **other useful techniques can be employed for anticipating features for game domains:**

- **Retrospection and trend analysis** [22]: in such a technique, the goal is to identify the ancestors of a given artifact (such as a game sample or feature) and understand how they evolved along the time. From this information, it may be possible to project future trends, new resources and functionality that can be built atop the current artifact state. The high availability of game samples (for instance, due to the "retro" branch of the game developer and player communities) makes this technique not only valid but encouraged.
- **Morphologic box** [23]: created by the Swiss astrophysicist Fritz Zwicky, this technique is a systematic form of idea finding where a solution for a problem is searched by trying out combinations from a matrix containing solution topics (or variables) and their possible values. The matrix is called a "morphologic box" and the combination of its values can result in unusual or even "weird" solutions, which are actually in tandem with the creativity element so essential to game development and design.

4 Case Study: The ArcadEx Game SPL

Following the guidelines presented in this paper, a game SPL was conceived, called ArcadEx. Its vision is presented below. Although such a vision is not enough to determine how every single possible generated product will look like, it provides a comprehensive, unambiguous high-level overview of what is expected from the SPL, with no

sole dependency on a game genre name. At the same time, it provides a baseline for refining the domain scope in next iterations and carrying out Domain Analysis.

- **ArcadEx Vision Statement:** The ArcadEx SPL is focused on generating uni- or multiplayer bi-dimensional arcade games for PC, with short levels composed by screens containing entities and walls, quick play action (in contrast to more in-depth gameplay or stronger storylines), simple, easy to grasp controllers, iconic characters and eventually rapidly increasing difficulty. Players control main characters who, or whose projectiles, collide with other entities such as non-player characters (NPC) or items. Victory condition is specified by the game designer as (a set of) game events: enemies are defeated, an object is collected, etc.
- **Target Platform:** PC (Windows).
- **Expectations for Core Game Dimensions**
    1. **Player:** single or local multiplayer mode is supported; each player controls one or more main characters.
    2. **Graphics:** Bi-dimensional world. Action screens display the world as viewed from above. Heads-up Displays (HUDs) based on progress bars, text, icons or radars can be used to display game or entity properties, such as health or time indicators.
    3. **Flow:** ArcadEx games are composed by a series of screens. A screen can display information or host actual game action. A screen can lead to and be reached from one or more screens.
    4. **Entities:** main characters are controlled by players. Other entity types are items and non-player characters. Entity attributes include position, velocity, direction and rotation. Animations (superposition of images at a given frame rate) are supported.
    5. **Events:** entities can be created or destroyed; collision detection; screen transition; changing an entity attribute value. Other events to be defined and refined by Domain Analysis.
    6. **Input:** keyboard and/or gamepad controller.
    7. **Audio:** sound effects supported as event reactions; background music can be associated with game screens and played in loop.
    8. **Physics:** collision detection, bouncing and some attraction forces. Screens can contain blocking walls.
    9. **Artificial Intelligence:** primitive AI concepts (e.g. path finding).
    10. **Networking:** ArcadEx games are standalone. There is no support for any kind of connectivity.
    11. **End-User Customization:** players will be able to edit the appearance of main characters (i.e., compose their frame images) through the use of a visual tool.
- **Negative Scope**
    1. **Physics:** no built-in support for elaborated physics models, such as fluids and friction.
    2. **Audio:** audio in ArcadEx will be as simple as playing background music and sound effects, without any built-in support to add special effect such as echo, 3D sound, etc.

3. **Graphics:** game screens of ArcadEx generated games will not support scrolling as a built-in feature. In other words, the boundaries of a screen will always be inside the dimensions of the game window. No built-in support for UI controls, such as menus, textboxes or drop-down lists.

For the ArcadEx SPL, about 30 games were selected and analyzed following the suggested guidelines, such as Pac-Man, Space Invaders, Asteroids, Defender, Geometry Wars, 1942, Missile Command and Rally-X, among others. About 2 to 4 man/hours were dispended in the analysis of each domain game. Guidelines were especially useful for discarding samples to analyze, filtering out conflicting features such as isometric (2D ½) views and more elaborated physics mechanisms. As the outcome of such Game Domain Analysis experience, a feature model with almost 150 features was built to describe the commonality and variability of the domain. Due to space constraints, only a subset of it (Flow concept) is displayed, in Figure 1.

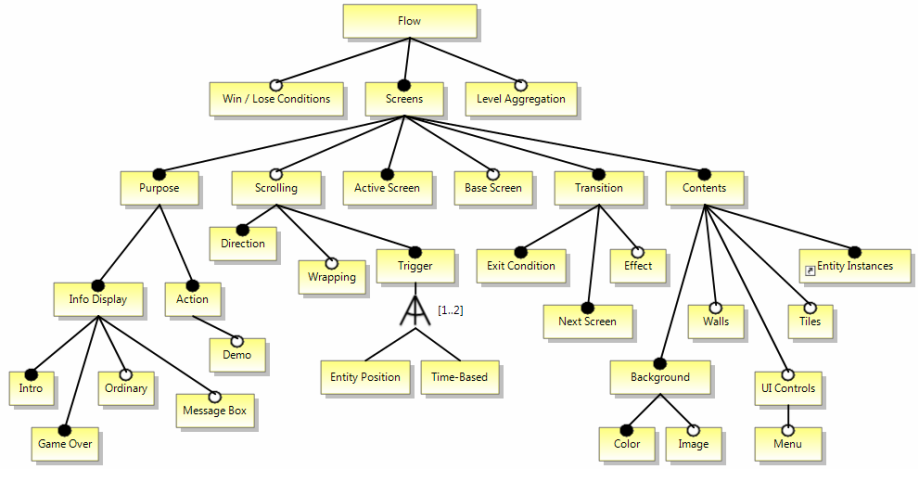


Fig. 1. Feature model subset for the ArcadEx game SPL: the Flow concept

5 Conclusions

Motivated by the peculiarities of digital games and their development process, this paper presented techniques for enriching Domain Analysis tasks targeted at digital games SPLs. Special considerations were given for tasks related to scoping the game domain, defining and refining game domain features, analyzing samples and anticipating features for a game domain. A case study for arcade-based games was used to illustrate and evaluate the proposed guidelines.

Some limitations of the presented work is that is does not constitute a complete Domain Analysis process per se, with a well-defined and comprehensive set of roles, tasks, inputs and outputs. Likewise, it does not attempt to comprehensively evaluate how current generic Domain Engineering tasks fit into digital games development.

Moreover, one important future work from here is conceiving controlled experiments to more formally validate the process

Although bridging the suggested Domain Analysis guidelines to Domain Design and Implementation is part of this research, such a discussion was left out of the scope of this paper. It consists in partitioning the game domain into sub-domains, assessing their automation potential and characterizing their variability (from routine configuration to creative construction). Then, sub-domains are prioritized and used as input towards SPL core assets such as DSLs and generators, compliant with a domain-specific game architecture. Such architecture is composed by fine-grained game components and game engines, which are ubiquitous in game development but now get promoted to "domain frameworks" through an adaptation layer.

As a concluding note, it is important to reemphasize that there is no such thing as a one-size-fits-all process. The proposed guidelines should be tailored to each game SPL reality, needs and constraints, in order to fully promote reuse and automation to the game development macro-domain.

References

1. Parnas, D.: On the Design and Development of Program Families. IEEE Transactions on Software Engineering (March 1976)
2. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading (2000)
3. Software Engineering Institute (SEI). Software Product Line Hall of Fame, `http://www.sei.cmu.edu/productlines/plp_hof.html` (retrieved on April 1, 2009)
4. Blow, J.: Game Development: Harder Than You Think. ACM Queue 1(10), 28–37 (2004)
5. Entertainment Software Association: Essential Facts about the Computer and Video Game Industry (2009)
6. Neward, T.: Why the Next Five Years Will Be About Languages. Keynote at the The ServerSide Java Symposium, March 27 (2008)
7. Greenfield, J., et al.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley & Sons, Chichester (2004)
8. Folmer, E.: Component Based Game Development: A Solution to Escalating Costs and Expanding Deadlines? In: Schmidt, H.W., Crnković, I., Heineman, G.T., Stafford, J.A. (eds.) CBSE 2007. LNCS, vol. 4608, pp. 66–73. Springer, Heidelberg (2007)
9. Reyno, E.M., Cubel, G.A.C.: Model-Driven Game Development: 2D Platform Game Prototyping. In: Game-On 2008, 9th International Conference on Intelligent Games and Simulation, pp. 5–7 (2008)
10. Rollings, A., Morris, D.: Game Architecture and Design. The Coriolis Group (2000)
11. Madeira, C.: FORGE V8: A Computer Games and Multimedia Applications Development Framework (in Portuguese), MSc dissertation, Federal University of Pernambuco (2003)
12. Neighbors, J.M.: Software Construction Using Components, Ph.D. Thesis, University of California (1980)
13. Prieto-Diaz, R.: Domain Analysis: An Introduction. ACM SIGSOFT Software Engineering Notes 15(02), 47–54 (1990)
14. Oxland, K.: Gameplay and Design. Pearson Education, London (2004)
15. Crawford, C.: The Art of Computer Game Design: Reflections Of A Master Game Designer. Osborne/McGraw-Hill, U.S (1984)

16. Anastasopoulos, M., Gacek, C.: Implementing Product Line Variabilities. In: Symposium on Software Reusability (SSR), Toronto, Canada, pp. 109–117 (2001)
17. Nascimento, L.M.: Core Assets Development in Software Product Lines: Towards a Practical Approach for the Mobile Game Domain. M.Sc dissertation, Federal University of Pernambuco, Recife, Pernambuco, Brazil (2008)
18. Rollings, A., Adams, E.: Fundamentals of Game Design. Prentice-Hall, Englewood Cliffs (2006)
19. Bonnie, R.: The Power of the Persona. The Pragmatic Marketer Magazine 5(4) (2007)
20. Koster, R.: A Theory of Fun for Game Design, Paraglyph (2004)
21. Kang, K., Cohe, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
22. Araujo, A.R.S.: Play4Fun: A Casual Digital Games Factory (in Portuguese) M.Sc dissertation, Federal University of Pernambuco (2009)
23. Zwicky, F.: Morphological Astronomy. The Observatory 68(845), 121–143 (1948)

Designing and Prototyping Dynamic Software Product Lines: Techniques and Guidelines

Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano

Centro de Investigación en Métodos de Producción de Software
Universidad Politécnica de Valencia
Camino de Vera s/n, E-46022, Spain
{ccetina,pginer,jjfons,pele}@dsic.upv.es

Abstract. Dynamic Software Product Lines (DSPL) encompass systems that are capable of modifying their own configuration with respect to changes in their operating environment by using run-time reconfigurations. A failure in these reconfigurations can directly impact the user experience since the reconfigurations are performed when the system is already under the users control. Prototyping DSPLs at an early development stage can help to pinpoint potential issues and optimize design. In this work, we identify and addresses two challenges associated with the involvement of human subjects in DSPL prototyping: enabling DSPL users to (1) trigger the run-time reconfigurations and to (2) understand the effects of the reconfigurations. These techniques have been applied with the participation of human subjects by means of a Smart Hotel case study which was deployed with real devices. The application of these techniques reveals DSPL-design issues with recovering from a failed reconfiguration or a reconfiguration triggered by mistake. To address these issues, we discuss some guidelines learned in the Smart Hotel case study.

1 Introduction

Software Product Line engineering has proven itself to be an efficient way for dealing with varying user needs and resource constraints. However, the focus has been on the efficient derivation of customized product variants that, once created, keep their properties throughout their lifetime. Previous research [12,8,1] shows that Dynamic Software Product Lines (DSPL) can assist a system to determine the steps that are necessary to reconfigure itself. Specifically, these systems can perform run-time reconfigurations in order to activate/deactivate their own features dynamically at run-time according to the fulfillment of context conditions.

For traditional Software Product Lines, once a product is obtained for a given configuration, it can be tested intensively before it reaches the users. However, the case of DSPLs is different since different configurations are obtained at run-time. A failure in DSPL reconfigurations directly impacts the user experience

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 331–345, 2010.

because the reconfiguration is performed when the system is already under the user control. Prototyping DSPLs at an early development stage can help to pinpoint potential issues and optimize design.

In this work, we identify and addresses two challenges associated with the involvement of human subjects in DSPL prototyping: enabling DSPL users to (1) trigger the run-time reconfigurations and to (2) understand the effects of the reconfigurations. On the one hand, reconfigurations are triggered by context events many of which are difficult to be reproduced in practice (e.g., the event of an unintended fire in the kitchen). To address this challenge, we have developed a technique that is based on RFID-enabled cards to easily specify the current DSPL context. On the other hand, when reconfigurations are performed, some of the effects are easily perceived (e.g., an alarm is triggered) while others are not (e.g., some sensors are deactivated). Thus, we consider that the direct observation of the DSPL system is not enough for evaluating the run-time reconfigurations. To address this challenge, we provide prototype's users with a configuration viewer tool which helps them to understand and evaluate the effects of the reconfigurations.

These techniques have been applied in the context of context aware smart spaces [5]. In particular, bu means of a Smart Hotel DSPL which was deployed with real devices. The Smart Hotel reconfigures its services according to changes in the surrounding context. A hotel room changes its features depending on users' activities to make their stay as pleasant as possible. Overall, the Smart Hotel comprises eight scenarios and eighteen reconfigurations among these scenarios. This DSPL was deployed in a scale environment with real devices to represent the Smart Hotel with human subjects using the prototype.

The application of these techniques reveals DSPL-design issues with recovering from a failed reconfiguration or a reconfiguration triggered by mistake. To address these issues, we discuss some guidelines learned in the Smart Hotel in connection to: (1) introducing user confirmations to reconfigurations, (2) improving reconfiguration feedback and (3) introducing rollback capabilities to reconfigurations.

In particular, the contribution of this paper is twofold as follows:

- The identification and solution of two challenges associated with the involvement of human subjects in DSPL prototyping: to (1) trigger run-time reconfigurations and to (2) understand the effects of the reconfigurations. Furthermore, theses techniques enable users to provide valuable feedback for DSPL redesign.
- The guidelines for addressing the following key issues of DSPL design: to (1) recover from a failed reconfiguration, and to (2) recover from a reconfiguration triggered by mistake.

The paper is organized as follows. Section 2 presents the running DSPL case studiy of the paper. Section 3 introduces the techniques for DSPL prototyping. Section 4 discusses the guidelines learned in the case study. Finally, Section 5 provides an overview of related work, and section 6 concludes the paper.

2 Case Study: The Smart Hotel DSPL

This section introduces the case study of a smart hotel, which reconfigures its services and devices according to changes in the surrounding context. The smart hotel was chosen as the reconfiguration-based case study for two main reasons: first, its nature as a shared environment in which different users use the same room over time. The clients each have their own preferences for the room, which should be adjusted to improve the quality of their stay; secondly, the preferences of the clients change depending on the activity performed (e.g., the clients usually have different preferences when they are watching a movie than when they are working).

Overall, the smart hotel case study describes the stay of one client in different scenarios. This includes the check-in process and the way the room interacts with the client and changes its features depending on the clients activities in order to make the stay as pleasant as possible. To give an idea of the dimensions of the case study, we present the following metrics:

According to the Feature Modelling technique, the Smart Hotel presents **thirty nine Features**. Some examples of these features are the Temperature Control feature, which offers a heating and cooling system; the Device Synchronization feature which synchronizes the devices that the user can have (e.g., laptop, mp3 player, or PDA) or the Security feature, which secures the room when the user is absent.

The main concepts of the Smart Hotel DSPL architecture are Services, Devices, and the Communication Channels among them. The Smart Hotel has **thirteen Services, twenty Devices and thirty-five Channels**. For instance, the Multimedia Service can establish communication channels to devices such as PDAs or MP3 players.

In the Smart Hotel, users can perform different activities. Specifically, our case study addresses **eight Scenarios**. These scenarios are: Check-in, Entering the Room, Working, Watching a Movie, Sleeping, Leaving the Room, Housekeeping and Check-out.

Detailed documentation about this case study is publicly available online at `http://www.carloscetina.com/papers/smart-hotel.eps`

2.1 The Run-Time Reconfigurations of the Smart Hotel DSPL

This section briefly presents the run-time reconfiguration Process of the Smart Hotel DSPL. The feature model specifies the possible configurations of the system, while the Dynamic Product Line Architecture can be rapidly retargeted to a specific configuration (see Fig. 1).

The first step of the Reconfiguration Process is to feed an ontology with context events. The context conditions check for values in this ontology. For instance, an *EmptyRoom* condition is fulfilled when none of the presence detection sensors is perceiving presence. This can be used to trigger the

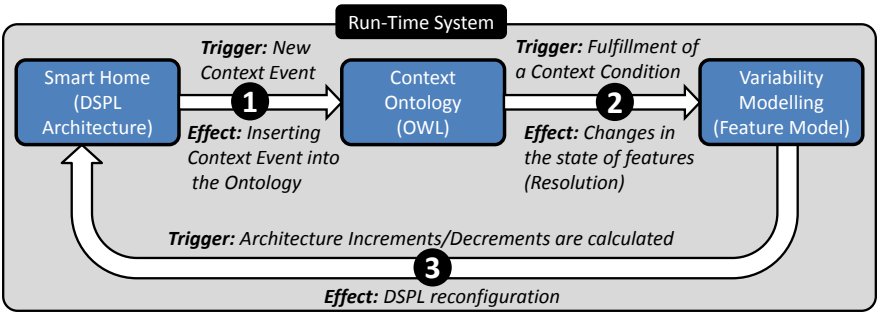


Fig. 1. Overview of the reconfiguration process

activation of both the *In Room Detection* and the *Occupancy Simulation* features when all the inhabitants leave room. We can also define another context condition, *Comfort*, to trigger the activation of features related to ease and well-being such as *LightingbyOccupancy* or *PipedMusic*.

The second step of the Reconfiguration Process is triggered when a context condition is fulfilled. Since a given condition can trigger the activation/deactivation of several features, we define the Resolution concept to represent the set of changes triggered by a condition. A *resolution* is a list of pairs where each pair is conformed by a Feature and the state of the feature. Each *resolution* is associated to a context condition and represents the change (in terms of feature activation/deactivation) produced in the system when the condition is fulfilled.

For instance, the $R_{EmptyRoom}$ resolution means that, when the DSPL senses that it is empty (condition), it must reconfigure itself to deactivate *Lighting by Presence* and to activate both *Presence Simulation* and *In Room Detection*.

The third step of the reconfiguration process (see Fig 1) addresses the architecture reconfiguration of the DSPL. In the $R_{EmptyRoom}$ example, the DSPL queries the Feature Model to determine the architecture for that specific context. The architecture increments and decrements are calculated in order to determine the actions that are necessary to modify the current configuration of the DSPL.

These increments and decrements indicate how system components should be reorganized for the reconfiguration in order to move from one configuration of the system (User in the room, see left side of Figure 2) to another configuration (Nobody in the room, see right side of Figure 2). As illustrated in Fig. 2, the presence sensors are no longer used for lighting (communication channels a and b are disabled), and they are used to provide information to the security service instead (communication channels e and f are enabled). In addition, the presence simulation service (labelled as 3) is activated, and the communication channels required for this service to communicate with multimedia (channel c) and lighting (channel d) are established.

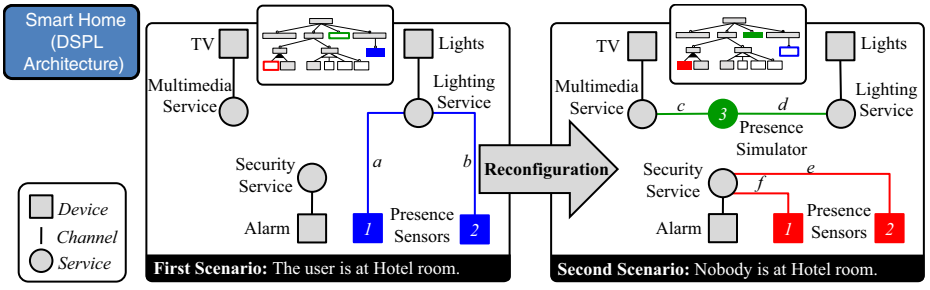


Fig. 2. Impact of active features on system components for two scenarios

3 Techniques for DSPL Prototyping

In this work, two major challenges were identified and addressed with the involvement of human subjects in the DSPL prototyping. These human subjects are potential users evaluating the prototype. DSPL reconfigurations are triggered by context events, many of which are difficult to reproduce in practice (e.g., a flood in the basement). To successfully prototype DSPLs, we must **enable users to trigger those reconfigurations that are relevant for the DSPL**, not only those reconfigurations that can be easily triggered.

When reconfigurations are performed some of the effects can be easily perceived (e.g., an alarm is triggered in the smart hotel) while others are not (e.g., some sensors are deactivated in the smart hotel). To successfully prototype DSPLs, we must **enable users to understand and evaluate the effects of reconfigurations**. If participants misunderstand reconfiguration effects, they will not be able to provide valuable feedback for system redesign.

3.1 Enabling Prototype's Users to Trigger Reconfigurations

Reconfigurations in the case study are triggered by different environmental conditions. When users are experimenting with the reconfiguration scenarios, they should be able to reproduce these situations in order to validate the system reaction. Since many context events are difficult to reproduce in practice (e.g., simultaneous events that occur in different rooms), simulating them is a must.

The control of context events is essential for the prototyping of DSPLs, since context changes are the events that drive the reconfiguration of the DSPL. Mechanisms should be provided to users to allow them to easily change the current context of the system. In this way, users can move from one configuration to another configuration by applying context changes.

In order to provide an intuitive representation of context events that users could manipulate easily, we provided them with cards that depicted these events. The use of the card metaphor was chosen since it is a familiar concept for most people [18].

Each *context card* represents a context event (such as "fire in the room"). During evaluation sessions, the users were given a deck of context cards. The deck

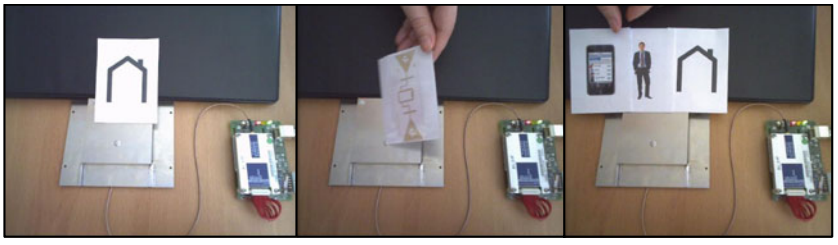


Fig. 3. Context Cards for triggering DSPL reconfigurations

included the events that could affect the particular DSPL being evaluated. The users could then make use of the context cards as the building blocks for triggering the reconfiguration of the DSPL.

The design of the context cards was driven by the elements defined in the Smart Hotel ontology. Each card involved a specific instantiation of a class from the ontology. The information provided in the card included the type element and, optionally, some relevant attributes regarding its particular instantiation (such as the location where the event takes place). When the cards were designed, we tried to avoid including too much information. Thus, the users could easily recognize the different cards at a glance (see Fig. 3, right).

In order to automate the evaluation process, the Context Cards were enhanced with RFID tags (see Fig. 3, left). When a card is placed on the table it is automatically detected by an RFID antenna, and the context ontology is updated accordingly. In this way, the cards can be easily manipulated as if it was part of a card game. Furthermore, they are also closely integrated with the DSPL reconfiguration engine. That is, setting a context card close to the RFID antenna triggered the different reconfigurations.

During the case study, the users could add and remove multiple cards from the table in order to define a specific context. The reconfiguration engine reconfigured the DSPL to fit the new context as it changed. Thus, the users could observe how the DSPL was reconfigured as they changed the context events.

The use of context cards enables users to evaluate the reaction of the system in different combinations of context events. Furthermore, putting users in control of the context definition provides valuable feedback for DSPL redesign. During our experimentation sessions, the users suggested new context cards and specific reconfigurations for certain context combinations that had not been previously considered by designers. Some new context cards were designed to group different events on a single card. Thus, a single card could represent the instantiation of several elements of the Smart Hotel ontology. This simplifies the activation of multiple conditions for users.

3.2 Enabling Prototype's Users to Evaluate the Reconfigurations

According to Dey in [3], one of the biggest challenges to the usability of context-aware applications (as is the case of a DSPL such as [12,9,19,13,15]) is the

difficulty that users have understanding why the applications do what they do. Dey defines the *intelligibility* concept as the support for users in understanding, or developing correct mental models of what a system is doing. This is done by providing explanations of why the system is taking a particular action and supporting users in predicting how the system might respond to a particular input.

Since the DSPLs that we are developing are context-dependant, intelligibility becomes a challenge for their evaluation. When the Smart Hotel is reconfigured, some of the consequences are easily perceivable by users (e.g., an alarm is triggered) while others are not (e.g., some sensors are deactivated). Thus, we considered that the direct observation of the physical devices by the user is not enough for evaluating the DSPL reconfigurations. Mechanisms are required by users to allow them to fully understand the reconfiguration consequences (e.g. changes that are produced in rooms where the user is not present, etc.).

For the evaluation process a Configuration Viewer has been developed to provide users with visual information about the reconfiguration effects in the system. This tool provides a graphical representation of the relevant entities in the Smart Hotel room. These entities include the devices, services, and communication channels among them. When a context condition is activated, it is also depicted in the Configuration Viewer. Thus, the user can easily perceive that motion sensors are enabled and provide information to the alarm system when the room becomes empty. Without the Configuration Viewer, users cannot be sure whether or not the presence detection has been turned on when they leave the room. As Fig. 4 shows, direct observation of the physical devices is not enough to evaluate run-time reconfigurations.

Since we are interested in the evaluation of DSPL reconfigurations, it is not enough to represent the Smart Hotel room in a single state. Therefore, complementary information is provided to the users through our tool to depict what has changed from the previous configurations. By clicking on services or devices, the users get detailed information indicating changes in the configuration (e.g., the motion sensors provide the user with the following message: "motion sensors no longer in use to control lighting, currently in use to control security.").

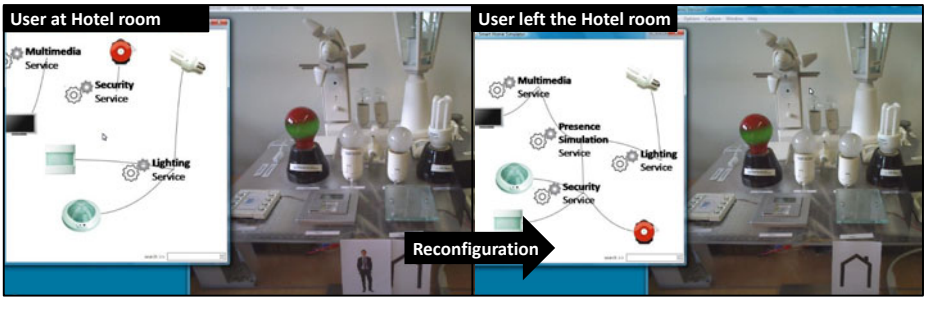


Fig. 4. Visualizing reconfiguration effects by means of the Configuration Viewer

This use of the Configuration Viewer enabled users to provide more accurate feedback for DSPL redesign during the Smart Hotel case study since they could determine what has actually changed.

3.3 Prototype Operation

In the prototype, a scale environment with real devices was used to represent the Smart Hotel. Therefore, the users could interact with the same devices that can be found in a real deployment (see Fig. 5, top-left). The Configuration Viewer was used during the experiments to keep track of the system evolution. This tool graphically depicts the devices, the services, and the connections among them that are present in the system at any given moment (see Fig. 5, bottom-left). Since the reconfigurations are performed as a response to context events, mechanisms are provided for triggering them. We adopted RFID cards to set the Smart Hotel context (see Fig. 5, right). Each of the cards symbolized context information such as the presence of users or the occurrence of different events. These cards were combined to insert events in the ontology and to trigger reconfigurations in the Smart Hotel.

During the experiment, the same user interaction with the environment (activating a presence detector) produced different results according to the current configuration of the system (which depended on the context expressed by the cards). For example, an initial scenario could consist of a room where one inhabitant is present. The cards that defined this scenario are the ones illustrated in Fig. 5. In this scenario, the system architecture was organized in such a way that the piped music was available and the presence sensors were used by the lighting service. The user of the prototype could listen to the music and the lights were turned on/off as the user interacted with the sensors. If the card that represented the hotel inhabitant was removed, the sensors were automatically no longer used for the purpose of light control but for security instead. As a consequence, when the user of the prototype interacted with the sensors again, the

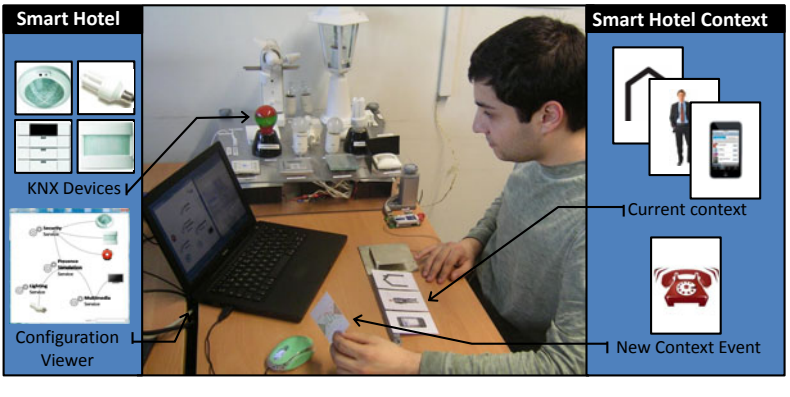


Fig. 5. Experimentation set-up

alarm went off (see this reconfiguration example and the prototype techniques online at http://www.autonomic-homes.com).

4 Guidelines for DSPL Design

Based on our experiences from these prototype techniques, we present the guidelines that we learned to assist researchers in the context of DSPL design.

4.1 Introducing User Confirmations to Reconfigurations

Using the Smart Hotel prototype, some subjects reported that they had triggered unintended reconfigurations by mistake (in opposition to a reconfiguration bug). In other words, they mistakenly set up the context for one reconfiguration scenario (i.e. *from EnteringTheRoom to LeavingTheRoom*), when they really wanted a different reconfiguration scenario (i.e., *from EnteringTheRoom to Working*). Unintended reconfigurations of this kind were not counted as DSPL failures since they were human mistakes. However, this behaviour raised an interesting point regarding whether or not a reconfiguration should be confirmed before its execution.

After analyzing the unintended reconfigurations performed in our case study, we realized that they can be classified into three different categories. These categories take into account the implications of returning to the source configuration. The three categories are the following:

Round-trip. If there is a reconfiguration that leads directly to the source configuration from the unintended configuration, then we classify the reconfiguration as a round-trip one (see Figure 6, left). In our case study, some subjects performed unintended round-trip reconfigurations between *EnteringTheRoom* and *LeavingTheRoom* configurations. For these unintended round-trip reconfigurations, the subjects did not require any special support since they could easily find the way to return to the source configuration. In fact, most of the reconfigurations were not reported as unintended ones in our case study, and those that were reported as unintended did not require support to find the way back. Based on this experience, we do not think that DSPLs should ask for user confirmation before performing a round-trip reconfiguration.

One-way. If there is no reconfiguration that leads directly (or indirectly) to the source configuration from the unintended configuration, then we classify the reconfiguration as a one-way one (see Fig. 6, center). In our case study, some of the subjects performed unintended one-way reconfigurations between the *LeavingTheRoom* and *Check-Out* configurations. For these unintended one-way reconfigurations, the subjects always required support since they could not find a way back to the source configuration. Based on this experience, we suggest that DSPLs should ask for user confirmation before performing a one-way reconfiguration. This suggestion comes from the fact that once a one-way reconfiguration has been performed, it is not possible to find a way back to the source configuration.

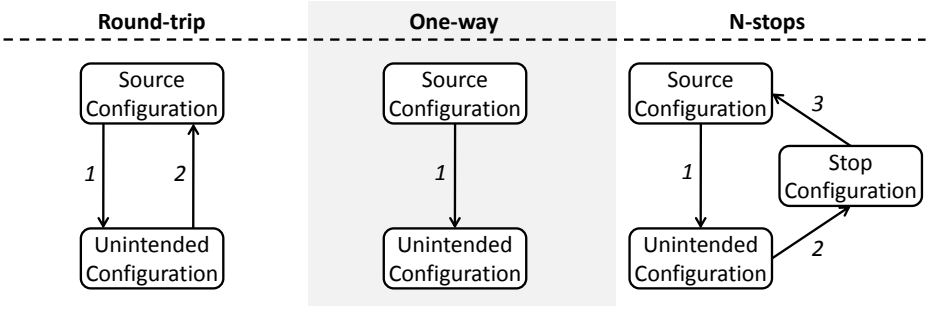


Fig. 6. Categories for confirmation of reconfigurations

N-stops. If there is a set of of reconfigurations that leads to the source configuration from the unintended configuration, then we classify the reconfiguration as a N-stops one (see Fig. 6, right). In our case study, some of the subjects performed unintended N-stop reconfigurations between the *EnteringTheRoom* and *Activity* configurations. For these unintended N-stops reconfigurations, almost all the subjects could easily find the way to return to the source configuration. However, a few subjects took a long time to find the way back. Based on this experience, we suggest that DSPLs should ask for user confirmation before performing an N-stops reconfiguration when the number of stops exceeds a certain limit. The purpose of our suggestion is to only require confirmation for critical reconfigurations. We also suggest identifying the acceptable limit of stops by applying Considerate Computing [6] techniques. These techniques take into account the domain particularities of the DSPL in order to determine when the number of reconfiguration stops is not trivial.

Since unintended reconfigurations can occur in DSPLs driven by context events [12,9,19,13,15] or by user actions [7], we believe that confirmation patterns defined in this study can help DSPLs engineers to mitigate the unintended reconfigurations. Furthermore, we think that these confirmation patterns are specially relevant for DSPLs driven by context events, since users of these DSPLs usually do not control all the feasible context events and can miss a specific configuration because of it. The confirmation guidelines that came from our case study experience can contribute to avoid this kind of undesired behaviour.

4.2 Improving Reconfiguration Feedback

When the users of the prototype perceived the effects of a specific reconfiguration, they sometimes noticed that the result was not the expected one. In those cases, they indicated the presence of a reconfiguration failure. One of the main issues with the identifications of these reconfiguration failures was related to the termination of the reconfigurations.

Since, each reconfiguration involves changes in different devices, services or communication channels, a delay between the event and the system reaction is introduced. This delay varies from reconfiguration to reconfiguration. Some subjects reported that it was difficult for them to determine whether the reconfiguration process was completed or there were still actions pending. This could lead to misidentifying failure or to misevaluating severity, since a subject could start evaluating a reconfiguration before it was actually finished.

To address this issue, our configuration viewer was enhanced with notification messages that indicated the completion of each reconfiguration. The subjects were provided with feedback regarding the overall process as well as at the service/device level. When a service or device was in the process of reconfiguration, it was depicted as busy (a waiting icon) in the configuration viewer.

Most of the subjects reported that they found this reconfiguration feedback to be very useful not only for failed reconfigurations but also for regular reconfigurations. Therefore, we suggest that DSPLs should provide feedback about the termination of reconfigurations, especially, when reconfigurations involve human users.

4.3 Introducing Rollback Capabilities to Reconfigurations

Our case study raised another important concern in connection with DSPL recovery after a failure. Once a reconfiguration was performed and evaluated using the prototype, a few subjects required support to resume the experimentation. They reported problems in performing the next reconfiguration after the failure. In other words, they did not find a simple way to reach another configuration of the case study. Below, we present the main kinds of issues reported and how we think they should be addressed in DSPLs.

Unexpected configurations. After a failure reconfiguration, a few subjects reported that the resulting configuration was not the expected one. In place of the expected configuration (i.e., *WatchingAMovie*), they got another configuration (i.e., *Working*). In most of these cases, the subjects could perform a new reconfiguration in order to reach the expected configuration. However, a few of the cases required several reconfigurations to reach the expected configuration. To address this issue, in DSPLs, we suggest introducing some sort of "undo" operation that returns the system directly to the previous configuration.

This has several implications for the design of DSPLs since some actions have collateral effects that cannot be easily undone (e.g., sending an e-mail). The handling of compensation actions to reverse a reconfiguration should be studied, also the consequences of a rollback need to be explained so that users can be provided information to help them choose among different compensation actions and understand how they relate to their desired goals.

Unknown configurations. After a failure reconfiguration, some subjects reported that they failed to identify the resulting configuration in the Smart Hotel documentation. In other words, the resulting configuration was different from all the documented configurations that made up the case study.

The Feature Model of the Smart Hotel defines more configurations than the ones considered in our case study. These *unknown configurations* imply that the subjects could not identify the set of reconfigurations that led to the expected configuration. Therefore, they needed support to continue the experimentation. To address this issue, we strongly suggest an "undo" operation that returns the system directly to the previous configuration. Note that for *Unknown configurations*, we think that the "undo" operation should be mandatory. However, for *Unexpected configurations*, we think that the "undo" operation should be optional since users have an alternative to achieve the expected configuration.

The DSPL that supports this case study makes use of Feature Models at run-time to determining how to perform the reconfigurations. According to a recent discussion on DSPL architectures [2], other DSPL approaches make use of different techniques to perform reconfigurations (i.e., QoS properties or UML profiles). Although the details are different, these DSPLs are based on variability specifications, and their reconfiguration can also lead to *Unexpected configurations* or *Unknown configurations*. Even though these DSPLs could achieve an expected configuration from any given *Unexpected* or *Unknown* configuration, our experience suggests that introducing an "undo reconfiguration" operation is simpler and more practical from the viewpoint of the DSPL user.

Finally, Table 1 summarizes the the guidelines presented in this section. On the one hand, left column shows the different kinds of reconfigurations identified

Table 1. Summary of reconfiguration kinds and their design implications

Reconfiguration	**Design Implications**
Round-trip Reconfiguration	This reconfiguration does not require to add *User Confirmation* since there is another reconfiguration that leads directly to the source configuration.
One-way Reconfiguration	This reconfiguration requires to add *User Confirmation* since since there is not another reconfiguration that leads directly (or indirectly) to the source configuration.
N-stops Reconfiguration	This reconfiguration requires the use of *Considerate Computing* to estimate the N which requires to add *User Confirmation*.
Not immediate Reconfiguration	This reconfigurations requires to add *visual feedback* about the reconfiguration status. We recommend that this feedback compromises both the time left and the scope of the reconfiguration.
Unexpected Reconfiguration	This reconfigurations does not require to add *roll-back capabilities* since the users can figure out another reconfiguration to reach the expected configurations. However, we recommend to add roll back capabilities when the context that drives the reconfigurations of the DSPL is not under the control of the users.
Unknown Reconfiguration	This reconfiguration requires to add *roll-back capabilities* since the users cannot figure out another reconfiguration to reach the expected configuration (some roll-backs may imply compensations).

by means of the prototype techniques. On the other hand right column shows the design implications for each reconfiguration kind.

5 Related Work

Since DSPL architectures are retargeted to different configurations at run-time, they could benefit from current approaches for adaptive architecture design. Specifically, Yacoub and Ammar [20] proposed a method for reliability risk assessment at the architecture level. This method is based on component-based systems in which implementation entities explicitly invoke each other. Liu et al. [14] identified architectural design patterns to build an adaptive architecture that is capable of preventing or recovering from failures. In comparison with our work, these methods do not address runtime reconfigurations that are driven by variability specifications such as Feature Models. However, we believe that some of techniques proposed in [20,14] (such as estimation of availability and severity) can also contribute to take DSPL prototyping one step further.

In the context of SPL evaluation, several approaches have produced results in connection to the design of SPL. For example: the F-SIG Feature-softgoal interdependency graph [11] and Zhang et al. [21] Bayesian Belief Network. There are also other methods that are not based on Feature Models such as COVAMOF (ConIPF Variability Modelling Framework) [16]. Most of these approaches usually remain at the Domain Engineering phase of SPLs only, they do not address run-time reconfigurations as our work does. Therefore, these approaches are not suitable for DSPL prototyping.

Other approaches address SPL products at run-time. The RAP approach [10] defines how some requirements should be mapped to the architecture and how the architecture should be analyzed in order to validate whether or not the requirements are met. Etxeberria et al. [4] also take into account SPL products at run-time . However, both the RAP and Etxeberria approaches are oriented to *static products* only. Conversely, our work addresses the evaluation of *reconfigurable products* such as these in DSPLs. Furthermore, we address the challenges of evaluating reconfigurable products and we provide guidelines to improve the development of future DSPLs.

6 Concluding Remarks

With more and more devices being added to our surroundings, simplicity becomes greatly appreciated by users. Dynamic Software Product Lines (DSPL) encompasses systems that are capable of modifying their own behavior with respect to changes in their operating environment by using run-time reconfigurations. However, failures in these reconfigurations directly impact the user experience since the reconfigurations are performed when the system is already under user control. This is in contrast to traditional SPLs where all the configurations are performed before delivering the system to the users.

Prototyping DSPLs at an early development stage can help to pinpoint potential issues and optimize design. To this end, we successfully identified and addressed two challenges associated with the involvement of human subjects in DSPL prototyping. On the one hand, DSPL reconfigurations are triggered by context events many of which are difficult to reproduce in practice. On the other hand, when reconfigurations are performed, some of the effects are easily perceived (e.g., an alarm is triggered) while others are not (e.g., some sensors are deactivated). If users misunderstand the reconfiguration effects, they will not be able to provide valuable feedback for DSPL redesign.

The above techniques have been applied with the participation of human subjects by means of a Smart Hotel DSPL which was deployed with real devices. In this case study, the hotel room changes its features depending on users' activities to make their stay as pleasant as possible. Since detailed documentation is publicly available online and the design of case studies is recognized as a difficult step [17], we believe that the Smart Hotel case study can be applied to more research in the context of DSPL.

The application of the prototyping techniques to the case study revealed DSPL-design issues with recovering from a failed reconfiguration or a reconfiguration triggered by mistake. To address these issues, we also discussed some guidelines learned in the Smart Hotel in connection to: (1) introducing user confirmations to reconfigurations, (2) improving reconfiguration feedback and (3) introducing rollback capabilities to reconfigurations. These guidelines enable DSPL engineers to analyse the DSPL reconfigurations in order to set up the DSPL platform regarding confirmations, feedback and rollback capabilities.

Finally, we conclude that the Smart Hotel DSPL achieved satisfactory results with regard to run-time reconfigurations; nevertheless, our prototype highlighted that DSPL engineers must provide users with more control over the reconfigurations or they will not be comfortable with DSPLs.

References

1. Cetina, C., Fons, J., Pelechano, V.: Applying Software Product Lines to Build Autonomic Pervasive Systems. In: 12th International Software Product Line Conference, SPLC 2008, September 8-12 (2008)
2. Cetina, C., Trinidad, P., Pelechano, V., Ruiz-Cortés, A.: An architectural discussion on dspl. In: 2nd International Workshop on Dynamic Software Product Line (DSPL 2008) (2008)
3. Dey, A.K.: Modeling and intelligibility in ambient environments. Journal of Ambient Intelligence and Smart Environments (JAISE) 1(1), 57–62 (2009)
4. Etxeberria, L., Sagardui, G.: Variability driven quality evaluation in software product lines. In: SPLC 2008: Proceedings of the 2008 12th International Software Product Line Conference, Washington, DC, USA, pp. 243–252 (2008)
5. Evesti, A., Eteläperä, M., Kiljander, J., Kuusijärvi, J., Purhonen, A., Stenudd, S.: Semantic information interoperability in smart spaces. In: Proceedings of the 8th International Conference on Mobile and Ubiquitous Multimedia, pp. 158–159 (2009)
6. Gibbs, W.W.: Considerate computing. Scientific American 292(1), 54–61 (2004)

7. Gomaa, H., Hussein, M.: Dynamic software reconfiguration in software product families. In: Software Product-Family Engineering, pp. 435–444 (2004)
8. Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. Computer 41(4), 93–95 (2008)
9. Hallsteinsen, S., Stav, E., Solberg, A., Floch, J.: Using product line techniques to build adaptive systems. In: 10th International Software Product Line Conference, 2006, August 21-24, 10 pages (2006)
10. Immonen, A.: A method for predicting reliability and availability at the architecture level. In: Software Product Lines, pp. 373–422 (2006)
11. Jarzabek, S., Yang, B., Yoeun, S.: Addressing quality attributes in domain analysis for product lines. IEE Proceedings Software 153(2), 61–73 (2006)
12. Lee, J., Kang, K.: A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In: 10th International Software Product Line Conference 2006 (2006)
13. Lemlouma, T., Layaida, N.: Context-aware adaptation for mobile devices. In: Proceedings IEEE International Conference on Mobile Data Management 2004 , pp. 106–111 (2004)
14. Liu, Y., Babar, M.A., Gorton, I.: Middleware architecture evaluation for dependable self-managing systems. In: Becker, S., Plasil, F., Reussner, R. (eds.) QoSA 2008. LNCS, vol. 5281, pp. 189–204. Springer, Heidelberg (2008)
15. Parra, C., Blanc, X., Duchien, L.: Context Awareness for Dynamic Service-Oriented Product Lines. In: 13th International Software Product Line Conference, SPLC 2009, pp. 24–28 (August 2009)
16. Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J.: Covamof: A framework for modeling variability in software product families. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 197–213. Springer, Heidelberg (2004)
17. Tichy, W.: Should computer scientists experiment more? Computer 31(5), 32–40 (1998)
18. Weal, M.J., Cruickshank, D., Michaelides, D.T., Howland, K., Fitzpatrick, G.: Supporting domain experts in creating pervasive experiences. In: Fifth Annual IEEE International Conference on Pervasive Computing and Communications, PerCom 2007. pp. 108–113 (March 2007)
19. White, J., Schmidt, D.C., Wuchner, E., Nechypurenko, A.: Automating product-line variant selection for mobile devices. In: 11th International Software Product Line Conference, SPLC 2007, September 10-14, pp. 129–140 (2007)
20. Yacoub, S.M., Ammar, H.H.: A methodology for architecture-level reliability risk analysis. IEEE Trans. Softw. Eng. 28(6), 529–547 (2002)
21. Zhang, H., Jarzabek, S., Yang, B.: Quality prediction and assessment for product lines, p. 1031 (2003)

A Software Product Line for the Mobile and Context-Aware Applications Domain*

Fabiana G. Marinho[1], Fabrício Lima[1], João B. Ferreira Filho[1], Lincoln Rocha[1], Marcio E.F. Maia[1], Saulo B. de Aguiar[1], Valéria L.L. Dantas[1], Windson Viana[1], Rossana M.C. Andrade[1], Eldânae Teixeira[2], and Cláudia Werner[2]

[1] Group of Computer Networks, Software Engineering and Systems (GREat), Computer Science Department (DC) Federal University of Ceará (UFC) Campus do Pici, Bloco 910, Zip Code 60455-760, Fortaleza, CE, Brazil {fabiana,marcio,valerialelli,lincoln,sauloaguiar,windson,bosco, fcofabricio,rossana}@great.ufc.br

[2] Software Reuse Group Systems Engineering and Computer Science Program (COPPE) Federal University of Rio de Janeiro (UFRJ) PO Box 68511, CEP 21945-970, Rio de Janeiro, RJ, Brazil {danny,werner}@cos.ufrj.br

Abstract. The mobile and context-aware application domain presents challenging requirements to software development. Although several solutions have been proposed for this type of application, reuse is not systematically used throughout the software development lifecycle. Then, in this paper we propose an approach for the development of a mobile and context-aware Software Product Line (SPL). A SPL for the mobile and context-ware mobile guide domain is presented in order to illustrate the steps of the proposed approach. Furthermore, the lessons learned in the SPL development are discussed. Both approach and SPL are the main contributions of this paper.

Keywords: context-awareness, mobility, software product line.

1 Introduction

The evolution of mobile computing has contributed to make information available everywhere and at any time. This evolution fostered the development of ubiquitous computing envisioned by Mark Weiser [21] and has allowed organizations to focus on software applications that consider user mobility, context-awareness and adaptability. This application domain, here named mobile and context-aware software, must hold adaptability as a principle. Mobile and context-aware software

* This work is a result of MobiLine project supported by CNPq (MCT/CNPq 15/2007 - Universal) under grant number 484523/2007-4. The main goal of MobiLine project is the construction of a mobile and context-aware SPL.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 346–360, 2010.

should be configured in order to be deployed on various device models. In addition, mobile and context-aware software should be able to adapt itself according to the changes in its context (e.g., users location, network conditions, etc.). Therefore, the adaptation mechanisms must be taken in account: i) during the design phase, in order to cope more easily with changes in the applications requirements and target environment; and ii) during the software execution phase, in order to be able to dynamically adjust its behavior dynamically following users needs, preferences, and current context.

In order to achieve adaptability, while requiring minimal user participation, mobile and context-aware software must use context information to provide services, interfaces or content tailored to the user's needs, expectations, and current situation [4]. Several solutions were proposed for development of mobile and context-aware software [11] [4] [17] [7] [16] [19]. Although most of them provide mechanisms that allow reuse, these solutions are not systematically used throughout the software development phases.

Software Product Lines (SPL) aim to construct software based on a family of applications, guiding organizations both on the development of new applications based on reusable artifacts (development with reuse), and on the construction of these artifacts (development for reuse) [13]. In [6] the UbiFEX-Notation for modeling features of context-aware product lines is proposed. We agree with [6] that approaches based on SPL can help the development of mobile and context-aware software in order to improve reusability and reconfigurability. However, like most existing approaches, UbiFEX-Notation is used only in the design phase, not supporting runtime adaptation, which is the problem addressed in this paper.

To investigate the viability of this assumption, a SPL for the mobile and context-aware application domain is proposed. It comprises three cycles. The first cycle identifies the commonalities present in mobile context-aware applications. The second cycle explores features presented in a specific sub domain. For this work, the Mobile Visit Guide sub domain was chosen, due to the existence of a large amount of applications, which permited the analysis of common requirements. Finally, in the last cycle, which corresponds to the SPL Application Engineering, the GREat Tour Mobile Guide, a product configuration of the proposed SPL is derived using core assets generated in the two previous cycles. Furthermore, an important contribution is a detailed description of how to build context-aware product lines.

The remainder of this paper is organized as follows: Section 2 presents related work; Section 3 describes the three cycles; Section 4 discusses the lessons learned; and finally, Section 5 presents conclusions and an outline of future work.

2 Related Work

The construction of SPL for the context-aware mobile domain is a relatively new area. Therefore, it is not clear how to cope with the fundamental complexity of this domain, such as context awareness, contingencies management and device heterogeneity [2]. Although there is still a lot to be accomplished in order to

fully understand this domain, ongoing works on SPL can be used to improve the reusability and reconfigurability of software products for this domain.

Lee and Kang [12] propose a SPL modeling approach based on binding time analyses of feature models. It consists in the identification of the association units and in determining the moment of association of the identified units. Furthermore, the Activation Rule concept is introduced, providing information concerning competition or mutual exclusion restrictions, dependency and priority, which must be considered during the activation of the association units. The authors state that it is possible to explicitly identify which functionalities can be associated with a product at execution time. However, the proposed approach does not define how to incorporate the concepts presented in the features model.

Van der Hoek [10] proposes the concept of "any-time variability", which involves the ability of a software artifact to vary its behavior at any point in the life cycle. This approach provides three functionalities: a variability representation; a tool to specify these variabilities; and tools to apply the results at different points of the life cycle. This work does not explore how context information is identified. Furthermore, the representation of variabilities has no way of identifying how this information influences the systems dynamic configuration.

Fernandes and Werner [6] propose the UbiFEX-Notation for feature modeling in context-aware product lines. It allows the explicit representation of entities and context information in a certain domain. It also represents the influence of this information in the product configuration. However, UbiFEX-Notation does not define how to implement these concepts for dynamic product reconfiguration.

Wagelaar [20] suggests the separation of internal interactions between characteristics and interactions caused by external factors. External factors are described in context models. The approach proposed by Wagelaar uses ontology to express the context and feature model. It also presented a mechanism to determine the validity of the configuration of a product for a given context.

Simons [18] states that in the process of developing a context-aware application, context model must contain not only information of type definitions, but also reflect meta information about the context, thus, the author proposes a UML profile called CMP (Context UML Profile).

Hartmann and Trew [9] proposes the variability context model that represents the variability of the product environment and is used to apply restrictions to the variability model, allowing the modeling of multiple product lines.

In SPL-based approaches, variability is typically defined at product configuration time, postponing decisions to be made at execution time. Our proposal considers the product variability as part of the feature modeling and the product configuration, detailing how to build a context-aware product line for the mobile application domain. Thus, the main contribution in the SPL proposed in this paper is to allow the creation of mobile and context-aware products that are reconfigurable and adaptable at runtime.

3 Software Product Line Development Process

The development of mobile and context-aware applications, considering the requirements of dynamic adaptation and reconfiguration can be accomplished more easily if carried out based on Service-Oriented Architecture (SOA). When using SOA, a large problem can be decomposed into smaller atomic parts, which facilitates the deployment, management, maintenance and evolution of mobile applications [14]. Mobile and context-aware applications based on SOA must be developed using mechanisms to find, access and assemble these small parts in a secure and fault-tolerant way. Therefore, these mechanisms must be incorporated into the SPL used to configure such applications.

The SPL scope for developing mobile and context-aware applications must identify the collection of applications that fit this domain and organize any relevant information using a feature model, aiming to identify commonalities and variabilities. Since this is an extensive domain, any attempt to build a general SPL for mobile and context-aware products would probably be inaccurate.

Considering this scenario, our proposal decomposes the mobile and context-aware domain into two different analysis levels that are referred throughout this paper as Base Level and Specific Level. Fig. 1 shows the Base and Specific Levels in terms of Domain Engineering and Application Engineering to configure a specific product.

The Base Level (Cycle 1) regards the features present in mobile and context-aware applications. The main characteristics identified were dynamic execution

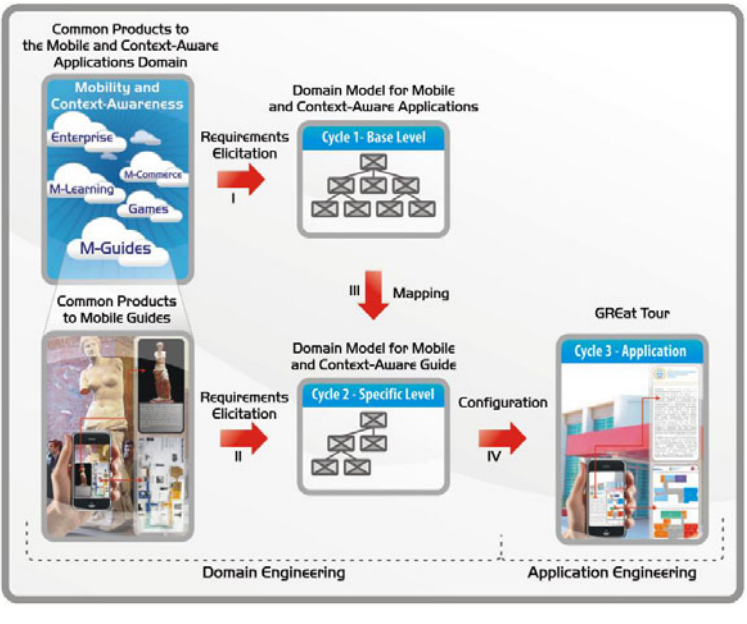


Fig. 1. Mobile Visit Guide SPL development process and a product configuration

environment, adaptability, and context-awareness, along with common features derived from the distributed nature of these applications, such as Message Exchange, and Service Description and Discovery [14]. These features were used as input for the Specific Level.

The Specific Level consists of the requirements present in a given business domain (Cycle 2). A second feature model is produced, merging the features required by the business domain with the features selected from the Base Level. The business domain chosen was the Mobile and Context-aware Visit Guide [1]. It comprises applications running on mobile devices to help visitors in an unknown environment (museums, parks, cities, etc.).

In the last cycle, a selection of the necessary components to configure a product is conducted by performing a clipping in the SPL for the Mobile Visit Guide. It reuses some core assets generated in the two previous cycles to configure an application. In this case, the product is a Mobile Visit Guide to the Computer Science Department at the Federal University of Ceará, called the GREat Tour.

3.1 Feature Model and Context Feature Model Notation

The feature models were created using the Odyssey-FEX [6] notation. It classifies the features into three dimensions: variability, optionality and category. Variability is divided into variation point, variant and invariant. Variation points represent points where decisions are taken. It can be configured by means of variants, that represent alternative features for configuring a variation point. Invariants are non-configurable domain features. An important concept is cardinality, which defines the minimum and maximum number of features that can be chosen.

Odyssey-FEX highlights the relationship semantics among features in the model, seeking to improve their representation and expressiveness. It combines Odyssey-FEX-specific relationships (Implemented By, Communication Link, and Alternative) with UML relations (inheritance, composition, aggregation, and association). These relationships are not only hierarchical, but they allow graphs to be formed, permiting the model expansion in several dimensions.

This paper uses the UbiFEX-Notation for context modeling [6]. Using UbiFEX-Notation, Context Entity is a feature used to represent context elements. Context information is an attribute of a context entity. Thus, Context Entities and their corresponding Context Information comprise the Context Definition, which is a situation that may trigger application adaptation. This adaptation is guided by the Context Rules.

3.2 SPL Cycle 1: Domain Requirements Engineering of Mobile and Context-Aware Applications

The SPL Cycle 1 identities commonalities and variabilities in mobile context-aware applications. This cycle generates the Base Level feature model, use cases and class diagrams, and any reusable components implemented in Cycle 1.

Table 1. Percentage of each feature in a total of 57 applications

	Features						
	Message Exchange	Mobility	Service Description	Service Discovery	Service Coordination	Security	Context Management
Percentage (%)	87%	100%	85%	85%	21%	70%	8%

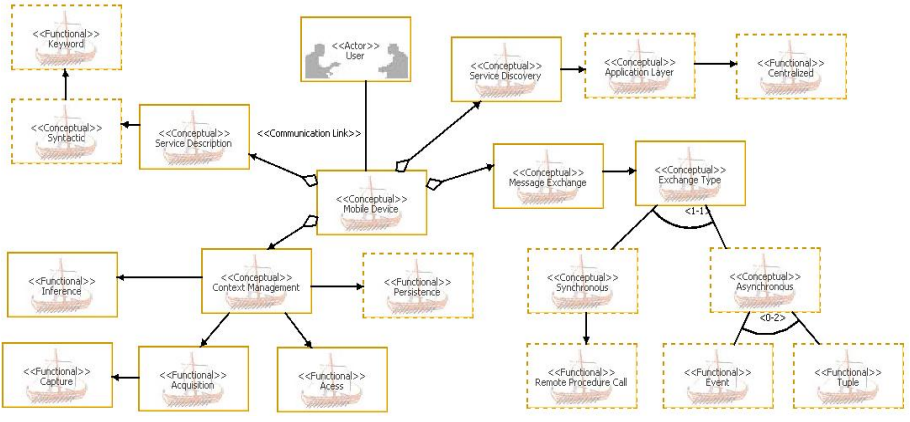


Fig. 2. Part of the feature model for Mobile and Context-Aware Applications

To identify these commonalities, a domain analysis was conducted using two separate approaches: i) review of a subset of published papers regarding context-aware mobile concepts. It identified six features for the Base Level: Message Exchange, Mobility, Service Description, Discovery and Coordination, as well as Security [14]; and ii) review of the applications developed in our Mobile Computing Research Group in the last five years[1].

Table 1 shows the presence of each feature in these applications. Mobility, Message Exchange, Service Description and Discovery were present in most of the applications and were modeled as mandatory. Although Context Management was rarely present(8%), it was modeled as mandatory, since our approach deals with context-aware applications. Finally, Security and Service Coordination were modeled as optional features. Fig. 2 shows part of this feature model. The complete model can be found at [15] and a detailed description of each feature can be found at Maia et al. [14].

Domain Design of Mobile and Context-Aware Applications. The architecture generated during the first cycle represents high-level conceptual features and their relationships (Fig. 3). To understand these relationships is important to define the way each functional feature is affected by other features. For instance, Fig. 3 shows non-functional features such as Adaptability, Context-Awareness,

[1] http://www.great.ufc.br/index.php?lang=en

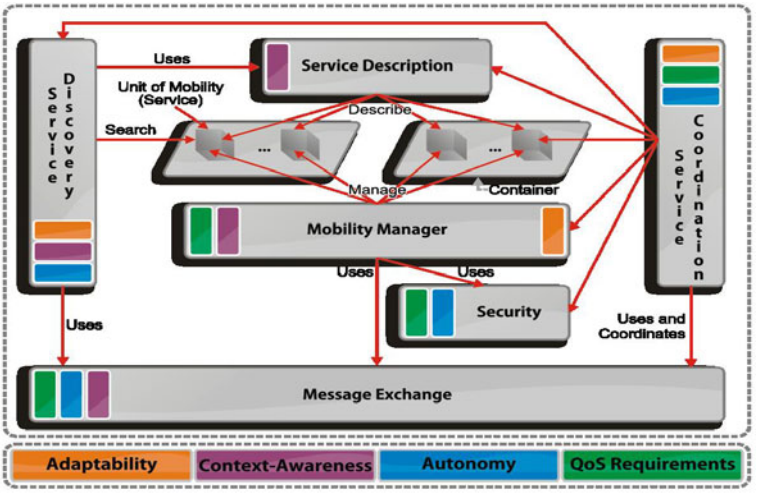


Fig. 3. Architecture for Mobile and Context-Aware Applications (Base Level)

Autonomy and Quality of Service affecting the functional features. With Odyssey-FEX, this relationship is modeled by using composition rules.

Domain Context Modeling of Mobile and Context-Aware Applications. The product configuration is highly dependent on context modeling, which is characterized by Context Entities and Context Rules. For instance, consider Network as being a Base Level Context Entity. It has five Context Information, namely Ontology Representation, Network type, Security, Latency and Server. Thus, for the first Context Rule, which is "if the Network Type assumes Cellular Network, it implies that the Service Discovery must use the Centralized feature". Another context rule is "if the Ontology Representation is present, it implies that the Service Discovery uses Description based on Keywords or Semantic". These Context Definitions and Rules are defined in tables (Table 2) and (Table 3). A complete set of Context Definitions and Context Rules can be found at [15].

3.3 SPL Cycle 2: Domain Requirements Engineering of Mobile Guide Applications

The Specific Level of the domain engineering comprises the requirements elicitation of a family of applications for a specific domain. The Mobile Visit Guide domain was used to specify the feature model for the Specific Level. Therefore, two actions were executed: i) identify the specific requirements of this sub domain, and ii) select relevant features from the Base Level generated at the first cycle.

Table 2. Examples of context definitions for Mobile and Context-Aware Applications

Expression Name	Context Information Feature	Relational Operator	Value
E1:Cellular Network	Network.Type	=	cellular
E2:Ontology Present	Network.Ontology Representation	=	true

Table 3. Examples of context rules for Mobile and Context-Aware Applications

R1: Cellular Network *implies* (Centralized)
R2: Ontology Present *implies* (Keywords OR Based on State)

In order to identify the features from the Specific Level, three surveys of Mobile Visit Guide were considered [1] [8] [5]. Tables 4 and 5 show the number of each feature present in a total of 15 Mobile Visit Guide analyzed.

Once specific features from the Mobile Visit Guide sub domain were identified, the features from the Base Level were configured and a complete feature model for Mobile and Context-Aware Visit Guides was created. The features selected from the Base Level were Context Management, Service Description and Discovery, Message Exchange, as well as Security. Fig. 4 shows part of the feature model for mobile and context-aware visit guides. The complete model can be found at [15].

According to Fig. 4, the actors that interact with the application are User, Item and Environment. Each Environment (Specific Level) has a Service Discovery mechanism (Base Level) to inform available services at a given moment (e.g. information about an Environment or Item). These services were described using a Service Discovery mechanism (Base Level). Additionally, this environment must use a Message Exchange mechanism (Base Level), based on Tuple space and Events.

To start the application, the User authenticates using a username and password provided by the Security feature (Base Level). Once authenticated, a user may visualize a map of the entire guide; choose to view information according

Table 4. Mobile Visit Guide features present in a total of 15 applications

View Map	View Location	View Environment	Capture User Profile	Capture Location	Permission Control
05/15	14/15	01/15	09/15	12/15	01/15

Table 5. Mobile Visit Guide features present in a total of 15 applications

Define Route	Authentication	View Environment Profile	Define Profile	List Items	View Item Profile
07/15	01/15	01/15	06/15	15/15	01/15

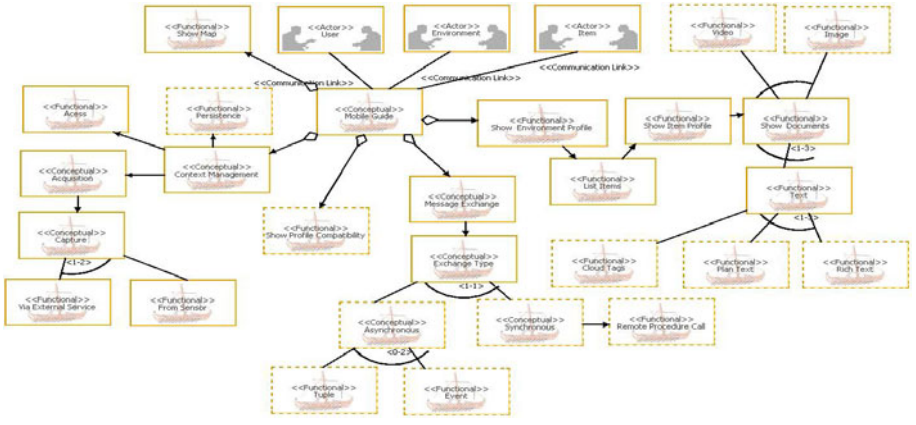


Fig. 4. Part of the feature model for Mobile and Context-Aware Visit

to his profile (Context Manager from the Specific Level) which was previously defined. Once an Environment is chosen, the user may select to view a list of items and choose to access information about a given item.

The Context Management is responsible for acquiring and providing information about the user indoor location. As a specific environment is entered, the user may obtain information about its location, items and the presence of other users. The behavior of the application is also affeted by device restrictions, such as remaining battery or available libraries.

Domain Context Modeling of Mobile Guide Applications. The clipping on the Base Level SPL carries its features, and Context Definition and Rules that affect the selected features. Once the features from the Specific Level are created, Context Definition and Rules for the Specific Level must be defined. For instance, the Context Entity Mobile Device has four Context Information, namely Memory, Libraries, Display and Battery. Thus, according to tables 6 and 7, "if the presence of Video Libraries in the Mobile Device assumes False, it implies that the feature based on Text and Image is used". The complete set of Context Definition and Rules can be found at [15].

3.4 SPL Cycle 3: GREat Tour Application Engineering

Application engineering uses the common assets available by the product line to create products. In this cycle, an application called GREat Tour was specified and implemented. These activities were distributed as 1) Application Requirements Engineering, 2) Application Design and 3) Application Context Modeling and Realization [10].

Application Requirements Engineering. GREat Tour is a tour guide for a research and development laboratory at the Federal University of Ceara. This

Table 6. Example of context definition for Mobile Visit Guide Applications

Expression Name	Context Information Feature	Relational Operator	Value
E1:Libraries Available	MobileDevice.Libraries	=	true

Table 7. Examples of context rule for Mobile and Context-Aware Applications

R1: NOT(Libraries Available) *implies* (Text AND Image)

application runs on the visitor's mobile device and provides information about the laboratory environment, researchers and environments that are visited. The behavior of these functionalities can be adapted according to the visitor current context, comprised by location, profile/preferences and device characteristics.

In order to configure the GREat Tour product, a clipping of the Mobile Visit Guide feature model was conducted. It is important to remark that the depicted model maintains variabilities that will be resolved only during the application runtime, according to the visitor's current context. The complete GREat Tour model is available at [15].

Application Design. In application design, an architecture that deals with the requirements of the product is derived from the reference architecture of the SPL Basic Level (Fig. 3). The major components and layers of the GREat Tour architecture are presented in Fig. 5. For example, the Message Exchange in this configuration is both Synchronous (SOAP for communicating with the Content Web Servers) and Asynchronous (communication with the Tuple Space). However, the event message model was not configured, since it was not necessary for this particular application.

Service Description, Discovery and Coordination are assured by the Tuple Space. The Context Management is distributed in the architecture. Context

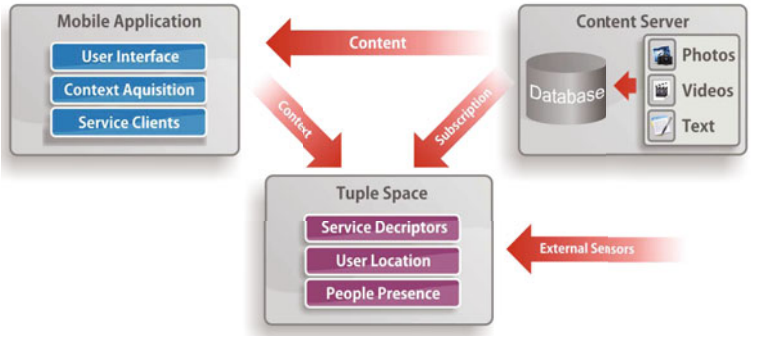


Fig. 5. GREat Tour Architecture Modules and Layers

Table 8. Examples of GREat Tour context definition

Expression Name	Context Information Feature	Relational Operator	Value
E1:Same Environment	Location Indoor	=	environment item
E2:Similar Profile	User Profile	≥	60%item profile

Table 9. Example of GREat Tour context rule

R3: Same Environment AND Similar Profile *implies* (List Items)

Acquisition services are deployed into the device and other context information is inserted on the Tuple Space by external sensors (for instance, those that inform people presence in a room).

Some variability points are still presented in this product, since they will be enabled or disabled according to the current context of the application. An example is the Context Acquisition, which may be aquired by sensors present in the mobile device or accessed from the Tuple Space.

Application Context Modeling and Realization. The behavior of the GREat Tour Mobile Guide is affedted by context information. For example, "the feature List Items is loaded automatically at runtime if the visitor has a profile consistent with the profile of an item in the present environment". Tables 8 and 9 show a subset of the Context Definition and Rules of the GREat Tour Mobile Guide. The complete set is available at [15].

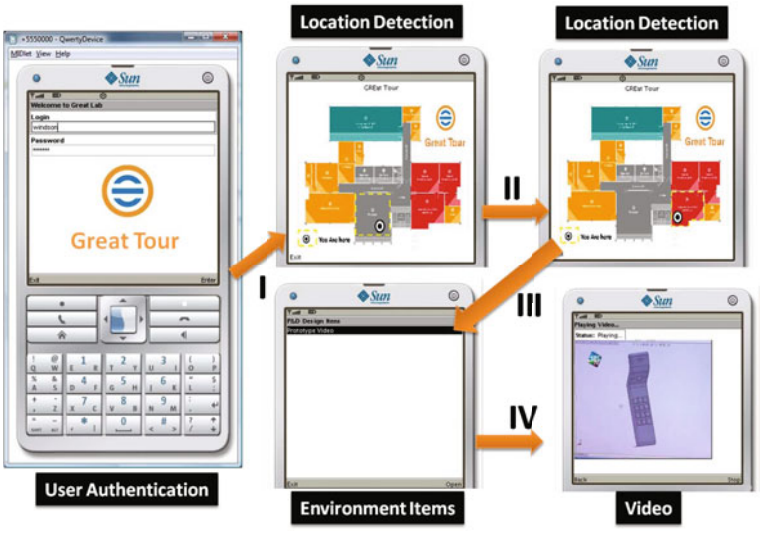


Fig. 6. GREat Tour executing in a mobile device emulator

Table 10. Main problems encountered in MobiLine SPL development

Classification	Problems Identified	Action Plan
Domain	– Generic SPL scope definition	– Study phase to identify the main concepts related to mobile and adaptive software development
	– Understanding the need for the development of generic SPL and specific SPL	– Requirements identification common to all mobile applications developed by members of the project
	– Specific SPL scope definition	– Study and documentation of requirements related to Mobile Guides development
	– Domain modeling	– Study the techniques of domain modeling – Participation in events related to the modeling and development of SPL
Theory	– Difficulty in understanding SPL concepts – Frequent changes and revisions of the models	– Workshops for discussion and exchange of experiences – Training among the members of the group
	– Traceability maintenance between the generated artifacts	– Allocation of experienced members in modeling and design – Use of the Odyssey Environment to help the mapping between the phases of the project development
Project Management	– Part time allocations	– Allocation of experienced members in order to avoid the need for new resources, such as M.Sc and Ph.D students
	– Effort above than expected to develop the generic SPL resulting from the requirements quantity and complexity	– Constant monitoring of project activities
Tools	– Odyssey Environment does not support different Java versions	– Maintain a single version of Java on machines used for modeling
	– Difficulty to visualize the composition rules documented in the models	– Elaboration of a document containing all rules included in the model
	– Impossibility to automatically configure the specific feature model from the generic feature model	– Use a copy of the generic feature model to allow the specific feature model development

Fig. 6 illustrates a few screenshots of GREat Tour Mobile Guide. The first user interface shows the realization of the Authentication feature. Once the user is logged, the Context Acquisition initiates. The system presents to the user its current location (I). When the user changes its environment position (II), the system shows another map indicating the new location. In this new room (after step II), rule R is satisfied and a List of Items is showed (step III). The Item is a Video describing the 3D mobile prototyping lab.

This application was developed in Java Mobile platform and it uses artifacts created during the firsts two cycles, such as the Tuple Space and the Context Management middleware. However, components and services were created specif-

ically for this application and they can be rewritten to a more reusable solution. The application tests were conducted using testing requirements for mobile devices proposed in [3].

4 Lessons Learned

MobiLine is a research project that aims at investigating characteristics existing in the development of mobile and context-aware software to build a software product line for this domain. During the process of developing this product line some difficulties had to be been overcome.

The problems identified are classified into four categories: Domain, Theory, Project Management and Tools. The Domain category is related to the complexity involved in the project, such as, mobility, context-awareness and adaptability, among others. The Theory category groups the problems faced in understanding and using the concepts of SPL. The Project Management category addresses the problems associated with the monitoring effort, and teams commitment and motivation. Besides, in the Tools category, difficulties encountered with the use of the adopted tools are described. The categories, the main identified problems and the actions taken are described in Table 10.

5 Conclusions and Future Work

This work presented a methodology for building mobile and context-aware SPL. Also, examples of modeling within the context of the MobiLine project were presented to illustrate the proposed approach.

The main contribution of this paper is a SPL for mobile and context-aware applications, along with process to build it. Therefore, the reader may find details and decisions that the authors faced, from the mobile and context-aware software domain engineering, all the way to the application configuration. Additionally, all artifacts that were generated are available either in the paper or in external links found throughout the paper. The results presented were obtained in the scope of the Mobiline project funded by Brazilian National Agency for Science and Technology (CNPq).

However, the proposed methodology only deals with the representation of concepts concerning context sensitivity in feature models, a model with a high level of abstraction. On the other hand, the architecture definition that serves as a basis for application instantiation within the same domain, representing the framework where architectural domain elements can be adapted or extended, is a key task in the construction process of a SPL. Accordingly, one of the main problems is the construction of an architecture that supports the adaptation dynamics and tracking between different levels of abstraction.

Hence, a future work is to study semantic mechanisms that exploit ontology and formal methods to help software engineers in the creation, specification and organization of architectural elements. The semantic augmentation of the feature model and context model is also being developed as a way to improve the integration of the different levels of abstraction.

References

1. Baus, J., Cheverst, K., Kray, C.: A survey of map-based mobile guides. In: Map-based Mobile Services, ch. 13, pp. 193–209. Springer, Heidelberg (2005)
2. Bronsted, J., Hansen, K.M., Ingstrup, M.: Service composition issues in pervasive computing. IEEE Pervasive Computing 9, 62–70 (2010)
3. Dantas, V.L.L., Marinho, F.G., da Costa, A.L., Andrade, R.M.C.: Testing requirements for mobile applications. In: 24th International Symposium on Computer and Information Sciences, ISCIS 2009, pp. 555–560 (14-16, 2009)
4. Dey, A.K.: Understanding and using context. Personal Ubiquitous Comput 5(1), 4–7 (2001)
5. Eisenhauer, M., Oppermann, R., Schmidt-Belz, B.: Mobile information systems for all. In: Proceedings of the Tenth International Conference on Human-Computer Interaction, vol. 4, pp. 354–358 (2003)
6. Fernandes, P., Werner, C.: Ubifex: Modeling context-aware software product lines. In: Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, Proceedings, Second Volume (Workshops), September 8-12, pp. 3–8. Lero Int. Science Centre, University of Limerick, Ireland (2008)
7. Grimm, R., Davis, J., Lemar, E., Macbeth, A., Swanson, S., Anderson, T., Bershad, B., Borriello, G., Gribble, S., Wetherall, D.: System support for pervasive applications. ACM Transactions on Computer Systems 22(4), 421–486 (2004)
8. Grün, C., Werthner, H., Pröll, B., Retschitzegger, W., Schwinger, W.: Assisting tourists on the move- an evaluation of mobile tourist guides. In: ICMB 2008: Proceedings of the 2008 7th International Conference on Mobile Business, pp. 171–180. IEEE Computer Society, Washington (2008)
9. Hartmann, H., Trew, T.: Using feature diagrams with context variability to model multiple product lines for software supply chains. In: SPLC 2008: Proceedings of the 2008 12th International Software Product Line Conference, pp. 12–21. IEEE Computer Society, Washington (2008)
10. van der Hoek, A.: Design-time product line architectures for any-time variability. Science of Computer Programming 53(3), 285–304 (2004)
11. Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, C., Campbell, R.H.: Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In: Coulson, G., Sventek, J. (eds.) Middleware 2000. LNCS, vol. 1795, pp. 121–143. Springer, Heidelberg (2000)
12. Lee, J., Kang, K.C.: A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In: SPLC 2006: Proceedings of the 10th International on Software Product Line Conference, pp. 131–140. IEEE Computer Society, Washington (2006)
13. Van der Linden, F.J., Schmid, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer, New York (2007)
14. Maia, M.E.F., Rocha, L.S., Andrade, R.M.C.: Requirements and challenges for building service-oriented pervasive middleware. In: ICPS 2009: Proceedings of the 2009 international conference on Pervasive services, pp. 93–102. ACM Press, New York (2009)
15. Mobiline - a software product line for the development of mobile and context-aware applications (March 2009), http://mobiline.great.ufc.br/index.php

16. Rocha, L.S., Castro, C.E.P.L., Machado, J., Andrade, R.M.C.: Using dynamic reconfiguration and context notification for ubiquitous software development. In: Proceedings of 21ht Brazilian Symposium on Software Engineering (SBES-XXI), pp. 219–235. SBC Press (2007) (in portuguese)
17. Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R.H., Nahrstedt, K.: A middleware infrastructure for active spaces. IEEE Pervasive Computing 1, 74–83 (2002)
18. Simons, C.: Cmp: A uml context modeling profile for mobile distributed systems. In: HICSS 2007: Proceedings of the 40th Annual Hawaii International Conference on System Sciences, p. 289b. IEEE Computer Society, Washington (2007)
19. Viana, W., Andrade, R.M.C.: Xmobile: A mb-uid environment for semi-automatic generation of adaptive applications for mobile devices. Journal of Systems and Software 81(3), 382–394 (2008)
20. Wagelaar, D.: Towards context-aware feature modelling using ontologies. In: MoDELS 2005 Workshop on MDD for Software Product Lines: Fact or Fiction? Montego Bay, Jamaica (October 2005) (position paper)
21. Weiser, M.: Some computer science issues in ubiquitous computing. Communications of the ACM 36(7), 75–84 (1993)

Using MDA for Integration of Heterogeneous Components in Software Supply Chains

Herman Hartmann[1], Mila Keren[2], Aart Matsinger[1],
Julia Rubin[2], Tim Trew[1], and Tali Yatzkar-Haham[2]

[1] Virage Logic, Eindhoven, The Netherlands
{herman.hartmann,aart.matsinger,tim.trew}@viragelogic.com
[2] IBM Research, Haifa, Israel
{Keren,mjulia,tali}@il.ibm.com

Abstract. Software product lines are increasingly built using components from specialized suppliers. A company that is in the middle of a supply chain has to integrate components from its suppliers and offer (partly configured) products to its customers. To cover the whole product line, it may be necessary for integrators to use components from different suppliers, partly offering the same feature set. This leads to a product line with alternative components, possibly using different mechanisms for interfacing, binding and variability, which commonly occurs in embedded software development.

In this paper, we describe a model-driven approach for automating the integration between various components that can generate a partially or fully configured variant, including glue between mismatched components. We analyze the consequences of using this approach in an industrial context, using a case study derived from an existing supply chain and describe the process and roles associated with this approach.

1 Introduction

Software product line engineering (SPLE) aims to create a portfolio of similar software systems in an efficient manner by using a shared set of software artifacts. SPLE is usually separated into two phases: domain engineering and application engineering, and a variability model is used to capture the commonality and variability and to configure a variant [1]. When implementing SPLE using model-driven architectures (MDA), it is conventional to create a domain-specific language (DSL) during domain engineering. During application engineering this DSL is used to create specific applications [2]. In component-based software development, a domain-specific component technology is used to create reusable artifacts, and a particular application is created by selecting components and binding variation points [3,4].

Due to the growing influence of software supply chains, an increasing portion of a product line is developed using commercial components [5]. In a supply chain, each of the participants uses components containing variability, combines them with in-house developed components, and delivers components containing variability to the next party in the supply chain [6,7].

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 361–376, 2010.

In an earlier paper we analyzed the consequences of integrating heterogeneous components in a software supply chain for resource constrained devices [8]. When software components from different suppliers have to be integrated, there may be mismatches between their interfaces, which have to be bridged by glue code. For instance, a set of interfaces might contain different numbers of methods (interface splitting), method parameters can be passed in different forms, e.g., as a struct vs. a list of separate parameters, methods having the same name might have different functionality implemented (functional splitting), etc.

A product line must be able to satisfy the requirements of its potential customers. Often, no single supplier can cover the full range of variability needed to achieve this, so it is frequently necessary to use components from several suppliers for a particular functional area. This leads to a product line that contains alternative components, only one of which can be used in a particular implementation [9], and a large number of glue components. In current practice there are three different approaches for integration and configuration of components, each with their limitations, as described below.

1. The possible glue components are created during domain engineering and configured during application engineering using a variability management tool. However, the manual creation of, possibly, a large number of glue components will require an unacceptably high development effort.
2. The required glue component is created during application engineering, at the moment that the specification of that glue component is known. This introduces an increase of throughput time which would be unacceptable in many situations.
3. A common standard interface and component technology is defined for a set of non-matching components. Glue components are created for each component to match these interfaces. Many components will therefore be bound through two glue components. This approach has the drawback that the glue components might become unnecessary complex if the standard interfaces have to cater for the interactions between any combination of components, thereby leading to additional development effort in comparison with the creation of custom glue.

To solve this problem we exploit the power of model-driven code generation to create custom glue between the combinations of supplied components that are actually used.

For resource constrained devices, some component technologies use static binding, rather than dynamic binding, and reachability analysis to exclude unnecessary code [3] and to create optimal system performance. The challenge that arises from a software supply chain is the ability to deliver a partly configured product to the next party in the supply chain. Components that have to be bridged may contain optional subcomponents. The glue components should only bridge between the components that are actually present so glue components should only be generated when the presence of those optional components is known. This can only be determined when the final configuration choices are known. In a supply chain, these final configuration choices could be made by a downstream participant. Furthermore, each supplier and the receiving parties may all use different build environments, which complicates the creation of the complete software stack. Other challenges from a supply chain relate to the protection of Intellectual Property and commercial interest, which means that, in many cases, the customer should neither receive source code, nor be aware of variation points that are offered to other customers.

We therefore address the following **research questions:**

1. Can MDA be used to bridge mismatches between components from alternative suppliers and can this method be used to support staged configuration? The set of mismatches we address in this paper is given in section 2.
2. What is the development process that is associated with this approach and what level of MDA expertise is required by the engineers?

Paper overview: In the paper, section 2 introduces a case study, Section 3 describes our approach and how it is supported with MDA and variability management tools. The management of the expertise expected of engineers is presented in section 4. Section 5 contains a discussion and identifies areas for future research, with Section 6 making a comparison with related art, followed by our conclusions in Section 7.

2 ZigBee Case Study

We demonstrate the applicability our approach by a realistic case study. In order to restrict the complexity of the first evaluation, the capabilities of the glue code in our tools are restricted to adapting between syntactic differences and the differences between component technologies. This caters for cases in which the semantics of interfaces are standardized, independently of the technology that may be used to implement them. As a case, we chose a heterogeneous ZigBee stack. ZigBee is a specification for a suite of high level communication protocols using small, low-power digital radios for wireless personal area networks [10]. The ZigBee stack is defined as a layered protocol (see Fig. 1). The standard is platform-independent, so implementers make their own choices for the exact form of the APIs between the layers. We focus on the lower layers, i.e. Physical, MAC, and Network. There is considerable variation between the network layers for different application profiles, e.g. "Plant Monitoring", "Home Automation" and "Smart Metering", so software suppliers usually only support a few such profiles. The MAC layer is independent of the application profiles, but has to be configured for the particular integrated circuit (IC). Therefore, in order to serve a range of customers, an IC vendor has to integrate ZigBee implementations from alternative suppliers, each of whom made their own software implementation technology choices (in our case nesC [11] and tmCom, a precursor to that used in the MPEG Multimedia Middleware standard [12]).

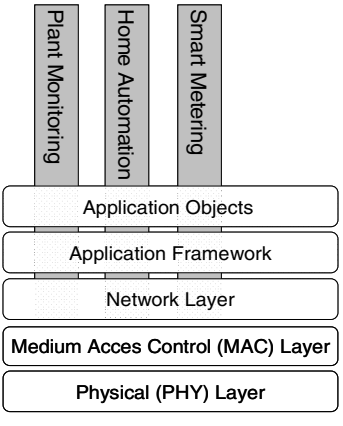


Fig. 1. Layers in the ZigBee protocol stack with profiles

The supply chain consists of IC vendors, software vendors, and the manufacturers that create the final product. In the supply chain, we found multiple parties for each link (typically more than five), each offering different sub-sets and extensions of the ZigBee standard. For each layer, a set of required, alternative and optional features is

specified, together with the dependencies between them. Examples of optional features are: power-saving, guaranteed time slot, and security mechanisms.

In this case study, we take the position of an IC vendor that is using software from specialized suppliers. We investigated the source code of three ZigBee stacks. Table 1 gives a subset of the features of these stacks, whose suppliers we will identify as A, B, and C. For a particular customer, the IC vendor selects the most suitable supplier and creates a partly configured product. For instance, a product can be configured with the components of Supplier C, where "Beaconing" and "Guaranteed Time Slot" are preconfigured, leaving "Power Saving" and "Security" to be configured by the customer.

Table 1. Feature set of selected suppliers

Feature\ Suppliers	**A**	**B**	**C**
Network layer	No	Opt	Opt
Beaconing	Opt	No	Opt
Guaranteed Time Slot	Opt	No	Opt
Security	No	Opt	Opt
Mesh-configuration	No	Man	No
Power Saving	No	No	Opt
Impl. Technology	nesC	C	C

In Table 1, *Man* stands for mandatory, *Opt* stands for optional, and *No* stands for not supported. We also listed the implementation technology. For the MAC layer, an Open-ZB implementation was used [13], based on nesC technology [11]. The differences between the technology choices of the Open-ZB implementation and those of the other suppliers were studied. In order to demonstrate the requirements for glue code generation, a dummy ZigBee Network layer was created, whose interfaces exhibited the union of the differences that had been found, thereby covering all the differences that we encountered in practice. The main differences between the technologies used in the Network component and the Open-ZB MAC are summarized in the Table 2.

Table 2. Differences between the technologies used in the ZigBee case study

	Network layer	Open-ZB MAC layer
Component technology	tmCom	nesC
Binding	Dynamic	Static
Interfaces	Uni-directional	Bi-directional
Naming convention	*tml*<port>*NXP*<interface>[*Ntf*] <method>	<port>_<interface> .<method>
Calling convention	Referencing structure	Individual parameters
Specific keywords	*STDCALL*	*command*, *event*

Both components have two ports (for data and control), which are connected correspondingly. To bridge the technology difference, an additional glue component must be added, as shown in Fig.2. The role of this glue component is to exhibit tmCom style interfaces toward the Network component and nesC style interfaces toward the MAC component. The glue component translates each down-call (*command*) that it receives from the Network component into a corresponding call to the MAC component. In this translation, it deals with naming and parameter-passing conventions. For the up-calls (*events*) originating from the MAC component, nesC uses static binding, based on a configuration description, whereas the tmCom Network layer employs

run-time binding, with a subscription pattern at the granularity of its interfaces [14]. Therefore the glue component implements the notification subscription management, which uses a table to map between the nesC functions called by the MAC and the tmCom functions in the Network layer. The glue code must provide the additional subscription management functions.

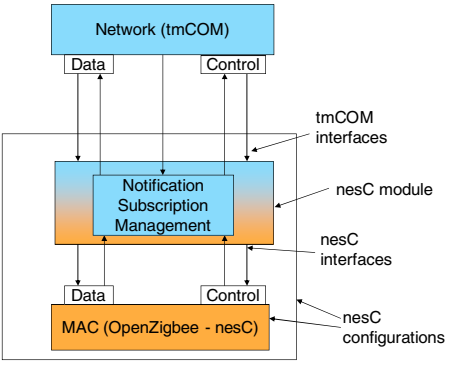


Fig. 2. Integration of Network and MAC layers

During development, staged configuration may be required. Once the supplier has been selected, the remaining choices for optional features shown in the table can either be made in-house or by the customer. For example, when Supplier C is selected, the optional feature "Power Saving" can be selected in-house and the choice for "Security" can be handed over to the customer. In this case, the customer receives code in which "Security" is still a variation point but "Power Saving" is already configured.

3 MDA for the Integration of Heterogeneous Components

This section begins with an overview of our approach and then elaborates on each step. The overview of our approach is described with respect to the transformations illustrated in Fig. 3.

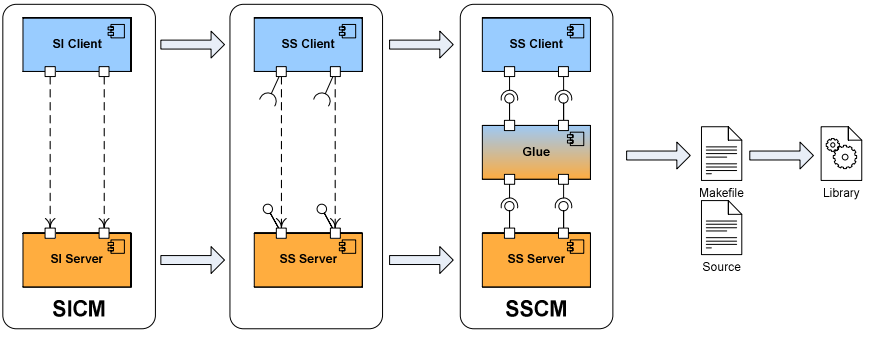


Fig. 3. Model transformations for integration and configuration of components

The presented model-driven approach uses the new variability pattern introduced in [8]. In this approach, an initial reference architectural model is defined containing conceptual components and their composition. These conceptual components represent architectural variation points, each of which describes the alternative suppliers of

that component. Later, this model is transformed into one in which the conceptual components are substituted by models of components from the selected suppliers.

We term the initial model a *supplier-independent component model* (SICM). This model, which is represented in terms of a new UML profile, consists of supplier-independent (SI) components and their dependencies. When the application engineer selects suppliers for the SI components, model-to-model transformations create a new model. In this new model, the SI components have been substituted by supplier-specific (SS) components that contain the interface descriptions for the corresponding development artifacts, which are tagged with their component technology.

Now, glue components are inserted by a second model-to-model transformation wherever a pair of incompatible interfaces is found. Then, a combination of model-to-code transformations and reusable code snippets is used to create the required glue code and other auxiliary files, such as build scripts, which can be transferred to the next participant in the supply chain.

To evaluate this approach, tool support for glue modeling was implemented as an extension to the IBM Rational toolset [15], including a new UML profile for modeling the supplier variability and glue specification, and model-to-code transformations for generating code artifacts. For feature modeling, a commercially available variability management tool was used [16], for which no extensions where needed. The process of using this approach, is illustrated in Fig. 4, and is further elaborated in the following subsections.

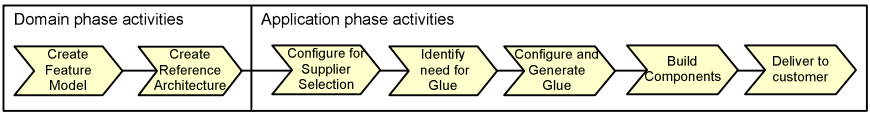


Fig. 4. Process description

3.1 Creation of a Feature Model and a Reference Architecture

As the first step in the process, the domain engineer defines the feature model that represents the product line variability, using a variability management tool. A distinction can be made between variation points that relate to product features, denoted as *functional variation points* (FVP), and variation points that describe alternative suppliers, denoted as *supplier variation points* (SVP), according to [9].

Then the SICM is created using the new UML profile. The SICM contains the SI components with their ports and variation points, as well as the dependencies between these components (left hand side of Fig. 3).

As potential component suppliers are identified during domain engineering, models of the SS components are defined. These components contain the interface descriptions for the supplied development artifacts. Subsequently, an *implement* connection with variability conditions is used to connect the SS components to their corresponding SI component, as shown in Fig. 5 for the ZigBee case study.

To complete the product line domain definition, the domain engineer activates validation rules to check that the SICM is legal and complete (e.g., each SI component must have a corresponding SS component to implement it).

The conceptual model of the entities used in this approach is shown in Fig. 6. It shows how the SS Components implements the SI components and their relation to the glue components. It also shows the relations between the components and the variation points (FVP, SVP). The different variability conditions for a particular SI component represent the SVP and are linked to the feature model in the variability management tool. The feature model is also linked to the FVPs of the SI components. Additionally, since different suppliers can implement the same FVP differently, and with different binding times, the SS components contain the configuration mappings of their FVPs to the FVPs of the SI components that they implement.

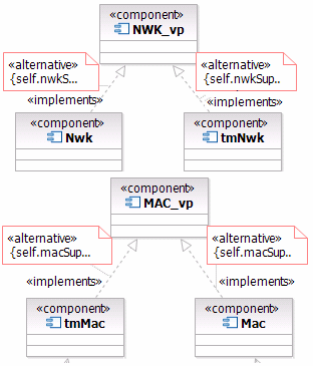


Fig. 5. SVP's of the case study

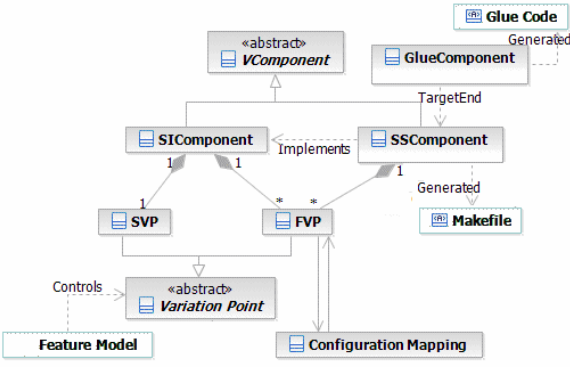


Fig. 6. Conceptual model of entities used

In the Zigbee case study the domain engineer modeled a subset of the ZigBee protocol architecture with two SI components, *NWK_vp* and *MAC_vp*, connected by two ports. Two alternative SS components are modeled for each SI component, *Nwk* and *tmNwk* for the *NWK_vp* SI component, and *tmMac* and *Mac* for the *MAC_vp* SI component (Fig. 5).

3.2 Configuration for Supplier Selection

During application engineering, an early step is the selection of the suppliers for each component, based on the different features that each supplier provides, together with non-functional criteria, such as cost. This choice is made using a variability management tool, which assigns values to the SVPs. A model-to-model transformation creates a new model in which any SI components with assigned SVPs are replaced by the selected SS component, corresponding to the middle model in Fig. 3. This transformation naturally supports staged configuration without any other measures being required to keep the partially configured feature and component models synchronized.

3.3 Identification of the Need for Glue

When resolving supplier variation points, the conceptual components of the initial model are replaced by the selected specific components, possibly from different suppliers. Here we consider the case in which alternative SS components associated with the same SI

component are implemented differently. Yet they should have similar functionality and equivalent ports. Glue components are required wherever connected ports have mismatched interfaces or where they have been implemented in different technologies. These conditions are detected automatically by validation of the UML model, based on the following criteria:

1. For each pair of SS components with connected ports, a *required* interface of an SS component does not match a *provided* interface, or any interface from which it inherits. Here, interface matching relates to the interfaces names and their method signatures (i.e. method names and the number, order and type of their parameters).
2. The components use different component technologies.

The model component elements are checked for the conditions above. Wherever the validation fails, a *glue* component is inserted between mismatched components by a model-to-model transformation, resulting in the right hand model in Fig. 3. At this point, the inserted component provides a specification for the glue, with its implementation still to be generated, as described in Section 3.4. Some component technologies, such as Koala [3] and nesC [11], require a top-level component to specify the composition of the components that use that technology, as illustrated in Fig. 2. Where required, the top-level component is also generated at this point.

When all SVPs have been resolved and all glue components have been added, we obtain the final, *supplier-specific component model* (SSCM).

3.4 Configure and Generate Glue Components

This section starts with a description of the meta-model for glue components. We then proceed with a description of tool support for the application engineer, who can interactively generate the glue code without being exposed to mechanisms of code generation, i.e. the model to text transformations.

Modeling of Glue Components

A glue component should resolve mismatches between interfaces, methods, method parameters and other mismatches between the glued components. Furthermore, it may also supply additional functionality that some component technologies may require. For example, in the case study, the nesC MAC expects that its notification interfaces will be bound statically at build time, whereas the tmCom NWK expects that the server provides a dynamic subscription management facility, which must now be provided by the glue. Other inconsistencies of supplier's implementation that must be resolved within the glue component are initialization, debugging, logging, and power management.

In order to create the glue components most efficiently during application engineering, their implementation is fully-generated from a model. This is in contrast to generating the skeleton of the code and completing it manually. The model combines information from the SSCM with parameterized code snippets created during domain engineering, and is configured interactively during application engineering using a set of wizards, which will be elaborated in the remainder of this section.

A meta-model for glue component models is defined in Fig. 7. Each glue component is associated with two SS components to be glued (the “TargetEnd” association), one of them may also be specified as "WrappedEnd' for cases where a top-level wrapper component is needed. The glue component contains ports according to the connected ports of the replaced SI components. For each such port it holds a number of interface maps. The glue component also has indicators for whether initialization and subscription management are to be included.

The relationships between the *provided* and *required* interfaces of the glue components are addressed at three levels of component integration: interface maps, method maps and parameter maps. Each kind of map has its specific attributes and snippets. This arrangement is able to support component integration even when different suppliers group methods into interfaces in different ways.

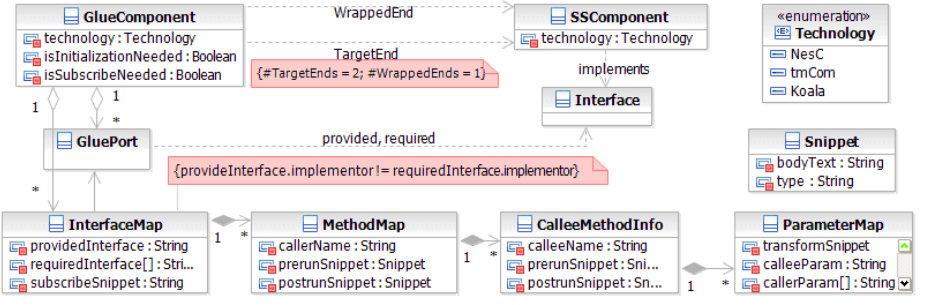


Fig. 7. Meta-model for a Glue Component

These relationships between interfaces are represented in the meta-model as follows. It contains an *InterfaceMap* for each interface provided by the glue. Each interface map contains a set of method maps that have one *caller* entity and a number of *callee* entities. This 1:n relationship caters for cases where the methods of the client and server are not matched, so that a single client call must result in a sequence of calls to the server. Each *callee* entity contains a set of parameter maps, which are used to control the transformations between those parameters that are passed in different forms by the *caller* and *callee* methods. Parameterized code snippet templates, stored in a library, are used for generating snippet instances inside of all these maps; these snippets are essential part of the generating glue code.

The code that is generated from the glue model consists of a set of methods for each *provided* interface of the glue component. The core of the *caller* method's body is a sequence of calls to the methods in the *CalleeMethodsInfo* list, together with the necessary parameter transformations. This sequence is contained within *prerun* and *postrun* code snippets, which can support other functionalities, such as memory management for temporary parameter structures or logging. When the application engineer has populated all the maps, a model-to-code transformation is used to generate the software artifacts, such as glue code, wrapper component, and build scripts.

Tool Support for Glue Code Generation:
To make the glue specification process faster with an effective use of the snippet templates tool support was developed on top of IBM RSA [15]. The implemented tool includes a set of wizards and dialogs to support the application engineer during glue configuration. These wizards hide the mechanics of code generation from the application engineer. For illustration, screen shots of the Interface Map Wizard, applied to the ZigBee case study, are shown in Fig. 8.

The Interface Map Wizard assists the user in specifying the mapping between provided and required interfaces that need gluing. Central part of the wizard is the Interface Map Editor, which allows to select one or more callee methods for each caller method, and to specify the transformations between the parameters of the methods. These transformations are captured as code snippets. The code for a snippet can be entered by the user either explicitly or by selecting a predefined snippet from a snippet library. The snippet library allows snippets to be reused across different interface maps and glue components. The snippet text can also be parameterized by predefined parameters such as interface name, callee or caller method name, etc. Parameter values can automatically be substituted by the model attributes taken from SSCM.

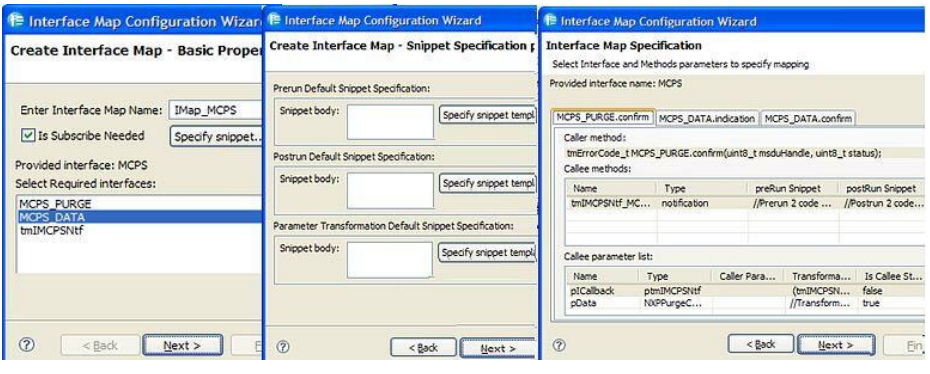


Fig. 8. Screenshots of the wizard for the Interface Map Configuration process

In addition, to reflect technology naming conventions, the tool allows usage of naming hints, which may significantly automate the method and parameter name matching process. We characterize the name's structure as a combination of the part that identifies the specific method or parameter, and additional parts (prefix, suffix and delimiters), e.g. the identifier of the interface, that together comprise the full identifier. In this way the additional parts may be stored for each supplier or technology, and used during the glue configuration process.

For the ZigBee case study we first created examples of glue methods manually, from which a library of parameterized code snippets was extracted. They are used later to configure numerous glue component maps in conformance with the meta-model described in Section 3.4. Code generation was implemented using the extensible JET-based Rational Software Architect transformation framework. A fragment of the code generated by our prototype is shown in Fig. 9.

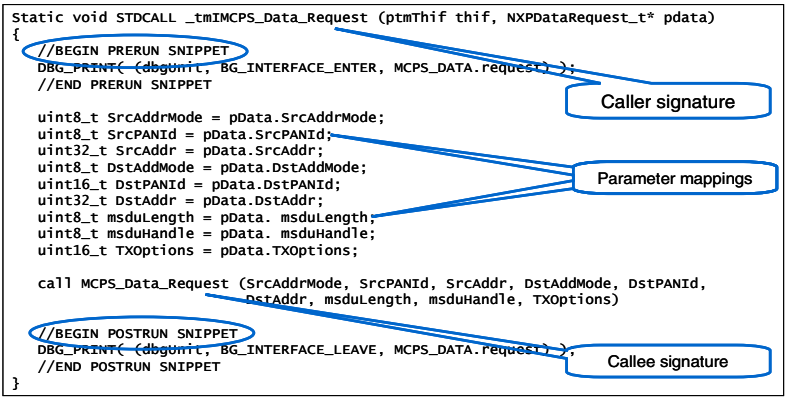


Fig. 9. Example of the generated glue code for the ZigBee case

3.5 Building the Components and Delivery to the Customer

Prior to the final build of the product, all the FVPs must have been configured, but we require flexibility in the configuration time for any FVP. The integrator makes an initial configuration, e.g. to protect intellectual property of other customers and suppliers and each customer receives a specialized configuration space containing only the remaining unconfigured variation points.

At the point that the code is validated and delivered, the mappings from the SI components' unconfigured FVPs to the corresponding variation points in the development artifacts are added to the generated build script. Subsequently, this mapping is used to translate the customer's configuration description. To address these two issues, we adopt a two-stage approach. For each programming language:

1. The components are passed through the early stages of the build process for their respective component technologies, to the point where standard language source and header files are generated. For example, Koala [3] identifies which source files will be required and generates macros to rename functions to permit static binding.
2. Having transformed all components to a standard form of source file for their language, build scripts are generated. These files include the FVP settings, such as the definition of pre-processor symbols used in the realization of variation points. Additional build scripts are generated for each glue component and a further, top-level build script identifies all the required components and validation is performed.

Where the customer only receives binary code, or where it is not possible to separate the two stages above, the final build is performed remotely on the supplier's site (e.g., by exposing the complete configuration and build process as a web service).

4 Development Roles

Given the small proportion of software developers who have experience with MDA technology, to be deployable in the short term, it is essential that only a few developers need to be familiar with the more esoteric aspects of MDA, such as defining UML

profiles and model transformations [2]. This section describes the development roles involved in the approach and the different levels of knowledge that each role requires in performing the activities, as illustrated in Fig. 4 and described in section 3.

The task Create Feature Model is performed by the *requirements manager* and requires a working knowledge of feature modeling. The task Create Reference Architecture is performed by the *domain architect*, who defines the product line architecture, represented by the SICM, and who also identifies the potential suppliers and creates the SS component models. This role requires a working knowledge of MDA.

The tasks Configure for Supplier Selection, Identify need for Glue and Configure and Generate Glue are performed by the *COTS engineer*. The *COTS engineer* is responsible for the integration of components from different suppliers and creating glue components. The *COTS engineer* should be familiar with the component technology and development environments used by the suppliers but he does not need specific knowledge of MDA since his tasks are assisted by the set of wizards that hide underlying MDA complexities. The tasks Build Components and Deliver to Customer are performed by the *customer support engineer*. The *customer support engineer* liaises with customers and determines what configuration is required prior to delivery. He will use the variability management tool to define a specific product configuration and uses the MDA tool to create the SSCM and to perform the final export. These tasks do not require knowledge of MDA principles. The *customer*, being the next link in the supply chain, requires no specific technical skills. He uses the variability management tool to make the final configuration of the received product artifacts, but is not exposed to any MDA technology.

As part of domain engineering team we recognize additional tasks, not described in Fig 4, to support the *domain architect*. The *COTS engineer* provides the requirements to the *transformation developer,* e.g. when a new component technology is used, who defines the transformations to generate the glue code and related artifacts. Here, the *transformation developer* requires very specific skills related to the transformation tooling. The *COTS engineer* has tasks during domain engineering as well as application engineering.

Finally, we identified the *Language Designer* [2] who is responsible for the definition of the meta-model and the model-to-model transformations. This work requires an in-depth knowledge of MDA. Since the meta-model and transformation can be reused for any component composition, these activities would typically be done by the MDA tool vendor.

5 Discussion and Further Research

The approach described in this paper allows components from different suppliers to be integrated, despite syntactic differences in interfaces and semantic differences related to component technologies that are based on a common programming language and variability mechanisms. It also supports staged configuration, where some variation points are resolved by the next participant in the supply chain.

The approach addresses the application engineering phase of SPLE, abstracting from the variability mechanisms used by each supplier and supporting the creation of glue components where they are required. It aims to make that glue generation as efficient as

possible, without making speculative investments during domain engineering. Once the glue snippet templates have been created for a few exemplars they are reused for numerous glue components.

The approach recognizes the limited experience of MDA available in the industry and restricts the number of development roles that need to be familiar with it. This is achieved by using a balance of reusable model-to-code transformations and parameterized code snippets, with a development environment that provides guidance to the applications engineer. Furthermore, our approach supports the export of standard programming languages and makefile technology, thereby avoiding exposure of the customer to unfamiliar technology.

The approach retains the sophisticated feature modeling techniques and supporting variability management tools developed for SPLE. For instance, these support staged configuration through the ability to present the engineer with a specialized configuration space, in which choices made at earlier stages are no longer accessible, although the constraints resulting from these choices are still in effect. However, the links to the development artifacts and, in particular, the mapping to the different variability mechanisms used by different suppliers, is now passed to the component models. The variability management tool is no longer required to determine where glue components will be required; this is now determined by a single model validation rule, providing a scalable solution. Hence, the variability model is now not *directly* connected to development artifacts [1], or model transformations [17], but the choice of the SVP now, *indirectly*, may lead to the generation of a glue component.

One of the challenges for further research is in the ability to deliver to customers partially-configured development artifacts while preserving the full capabilities of component technologies, such as nesC [11] and Koala [3], that minimize the memory requirements of the code through their reachability analyses of their components. The current approach converts all components into standard source files and generates a uniform style of build script, which propagates the remaining FVPs, thereby avoiding the customer from being confronted with multiple build environments. Currently, the creation of standard source files uses the native build process for each component technology. However, the build process for some component models, such as nesC [11], only creates conventional C files once the C pre-processor has been run, by which time all variation points with design-time binding will have been instantiated. However, by developing new build environments that are aware of variation points, both staged configuration and reachability analysis can be supported for these models.

A principal area for further research is the bridging of greater semantic gaps between components. This would allow the approach to be applicable to a much larger range of glue code generation. The current approach supports moderate mismatches because of the ability to map one *caller* function to a sequence of *callee* functions. This is sufficient for the ZigBee case, whose standard defines the semantics of messages *within* the communication stack, but there are many standards for embedded products that only consider the interaction between the product and its environment, with no consideration for the APIs of the software within the product. Egyed and Balzer [18] have proposed a reference architecture for stateful glue components, which may form the basis of more capable glue components for COTS integration. Here further research is required to extend the automated support to be able to guide creation of this style of wrapper. A related issue is that the current approach only inserts glue

components between pairs of components. However, there are cases, such as the generation of code to map from the operating system abstraction layers (OSAL) of each supplier to the actual OS, that cannot be done in a pairwise manner, because of the need for common book-keeping for shared OS resources. Therefore, a more general model of glue code must be developed for these cases.

6 Comparison with Related Art

From the perspective of staged configuration in software supply chains [6], we address organizations in the middle of the chain, which must both integrate components from different sources and pass partially-configured artifacts on to downstream customers. The problem of the use of feature models for coordinating the configuration of artifacts using different variability mechanisms is addressed by Reiser *et al.* [19]. However, they do not consider the creation of glue components. We have previously discussed merging feature models from alternative suppliers for a particular feature area [14], but that paper did not consider how glue components would be addressed.

Gomaa [20] addressed the use of UML to represent feature models. While doing so would have resulted in only a single tool being used, UML must be heavily profiled for this purpose.

Voelter and Groher describe the integration of a variability management tool with a model-based software development environment [17]. However, they address the links to transformations for in-house developed components, rather than the needs of a software supply chain.

The definition of the SICM for the ZigBee case study was straightforward, given the reference model in the standard. Where there is no pre-existing reference mode, the *architectural reconciliation* approach, proposed by Avgeriou *et al.* [21], to defining a COTS-based architecture can be used. However, while their approach aims to avoid architectures that require excessive amounts of glue code, they do not address how the essential glue code would be created efficiently.

Zhao *et al.* [22] address the combinatorial explosion of potential glue components when bridging between different component technologies. They use a generic grammar to specify the implementation of glue, but they avoid having to handle hybrid build processes by using SOAP as a common communication format between all technology types. This is approach is unacceptable for resource-constrained devices, in which code size and performance remain critical. Smeda *et al.* [23] address the creation of the specification of glue components from the composition of parameterized templates in the context of an architectural description language and address the creation of a modeling tool for this language. However, they do not address how automated support could be given to developers to assist in template composition.

Stahl *et al.* [2] and Krahn *et al* [24] describe the different roles needed in MDA and the skills required. Where they provide a classification for the roles during domain engineering, we additionally provide the different roles and skills, associated with our approach, during application engineering and for the customer's organization.

7 Conclusions

In this paper, we presented a model-driven approach for automating the integration of heterogeneous components from different suppliers, covering syntactic mismatches and semantic mismatches related to different component technologies. We exercised our approach on a case study that is derived from an existing supply chain, for which we used a commercially available variability management tool [16], and a prototype was implemented as an extension to IBM Rational MDA tool [15]. We described the process and roles that are associated with our approach.

In this paper we showed that the approach has the following benefits compared to prior art and current practice:

- Glue components are generated efficiently only when they are required, thereby avoiding unnecessary development effort during domain engineering.
- Staged configuration is supported; offering the next party in the chain to do the final configuration, while providing a route to preserving the capabilities of component technologies in this domain to minimize code size.
- The additional skills required to deploy MDA are localized in the organization by providing tool support for configuration and glue code generation, which ensures that only a limited group of developers are exposed to unfamiliar technology.

Finally, we identified how our approach can be extended to support specialized component technologies and to bridge greater semantic differences.

References

1. Pohl, K., Bockle, G., van der Linden, F.: Software Product Line Engineering. Springer, Heidelberg (2005)
2. Stahl, T., Voelter, M.: Model-Driven Software Development. Wiley, Chichester (2005)
3. van Ommering, R.: Building Product Populations with Software Components. PhD. Rijksuniversiteit Groningen (2004)
4. Atkinson, C., et al.: Component Based Product Line Engineering with UML. Addison-Wesley, Reading (2002)
5. Wallnau, K., Hissam, S., Seacord, R.: Building Systems from Commercial Components. Addison-Wesley, Reading (2002)
6. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration through Specialization and Multi-Level Configuration of Feature Models. Software Process Improvement and Practice 10, 143–169 (2005)
7. Hartmann, H., Trew, T.: Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains. In: 12th International Software Product Line Conference (2008)
8. Hartmann, H., Keren, M., Matsinger, A., Rubin, J., Trew, T., Yatzkar-Haham, T.: Integrating Heterogenous Components in Software Supply Chains. To be published in 1st ICSE workshop on Product Line Approaches in Software Engineering (2010)
9. Hartmann, H., Trew, T., Matsinger, A.: Supplier Independent Feature Modeling. In: 13th International Software Product Line Conference (2009)
10. ZigBee Alliance, `http://www.zigbee.org/`

11. Gay, D., Levis, P., van Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC Language: A Holistic Approach to Networked Embedded Systems. In: Conference on Programming Language Design and Implementation ACM 2003 (2003)
12. ISO/IEC 23004-3:2007, Information Technology – Multimedia Middleware – Part 3: Component Model. International Organization for Standardization (2007)
13. Cunha, A., Koubaa, A., Severino, R., Alves, M.: An Open-Source Implementation of the IEEE 802.15.4/ZigBee Protocol Stack on TinyOS. Polytechnic Institute of Porto (2007)
14. ISO/IEC 23004-1:2007, Information Technology – Multimedia Middleware – Part 1: Architecture. International Organization for Standardization (2007)
15. IBM Rational Software Architect for WebSphere software, `http://www-01.ibm.com/software/awdtools/swarchitect/websphere/`
16. Pure::Variants, Variability Management Tool, `http://www.pure-systems.com`
17. Voelter, M., Groher, I.: Handling Variability in Model Transformations and Generators. In: 7th OOPSLA Workshop on Domain-Specific Modeling (2007)
18. Egyed, A., Balzer, R.: Integrating COTS Software into Systems through Instrumentation and Reasoning. Automated Software Engineering 13, 41–64 (2006)
19. Reiser, M., Tavakoli Kolagari, R., Weber, M.: Unified Feature Modeling as a Basis for Managing Complex System Families. In: 1st International Workshop on Variability Modeling of Software-intensive Systems (2007)
20. Gomaa, H.: Designing Software Product Lines with UML. Addison-Wesley, Reading (2005)
21. Avergiou, P., Guelfi, N.: Resolving Architectural Mismatches of COTS through Architectural Reconciliation. In: Franch, X., Port, D. (eds.) ICCBSS 2005. LNCS, vol. 3412, pp. 248–257. Springer, Heidelberg (2005)
22. Zhao, W., Bryant, B., Burt, C., Raje, R., Olson, A., Auguston, M.: Automated Glue/Wrapper Code Generation in Integration of Distributed and Heterogeneous Software Components. In: 8th IEEE International Enterprise Distributed Object Computing Conference (2004)
23. Smeda, A., Oussalah, M., ElHouni, A., Fgee, E.-B.: COSABuilder: an Extensible Tool for Architectural Description. In: 3rd International Conference on Information and Communication Technologies (2008)
24. Krahn, H., Rumpe, B., Völkel, S.: Roles in Software Development using Domain Specific Modeling. In: 6th OOPSLA Workshop on Domain-Specific Modeling (2006)

Mapping Features to Reusable Components: A Problem Frames-Based Approach*

Tung M. Dao and Kyo C. Kang

The Department of Computer Science and Engineering,
Pohang University of Science and Technology (POSTECH), Pohang, Korea
{tungdm,kck}@postech.ac.kr

Abstract. In software product line engineering (SPLE), feature modeling has been extensively used to represent commonality and variability between the products of a domain in terms of features, based on which reusable components are developed. However, the link between a feature model and product requirements, that fundamentally decide how the features are developed into reusable components, has not been adequately addressed in SPLE methods. This paper introduces an approach to combining feature modeling and problem frames in an attempt to address this problem. First, features are mapped to problem frames using heuristics derived from feature modeling and feature mapping units. Requirements are then identified and analyzed to ensure that they are fully satisfied. Finally, a solution modeling method maps the problem frames to architectural components. A Home Integration System (HIS) case study is used to demonstrate the feasibility of the approach.

Keywords: feature model, problem frames, reusable components, software product line.

1 Introduction

Software product line engineering (SPLE) is an emerging paradigm that helps organizations develop their software products from core assets (e.g., reusable components) [1]. Instead of developing a single system from scratch, SPLE allows organizations to develop a family of software systems that share common and variable *features*. Consequently, feature models play an essential role in SPLE. Since their initial introduction in FODA [2] nearly two decades ago, many product line methods have been suggested, including Generative Programming (GP), FOPLE, FORM, and FeatuRSEB, in which feature models are systematically and extensively applied [3]. In the FORM method [4], for example, during the domain engineering step, a feature model is generated as an output of domain analysis. The feature model is then used to derive a reference architecture and develop reusable components.

* This research was supported by the National IT Industry Promotion Agency (NIPA) under the program of Software Engineering Technologies Development.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 377–392, 2010.

Ultimately, the derivation process is a transformation from common and varying requirements in the problem space to reusable components in the solution space of a product line. Thus, when a feature is developed into a reusable component, it is critical to systematically identify, analyze, and understand all of the requirements as well as the potential concerns or issues related to that feature. This implies that the link between the two concepts of *features* and *requirements* in software product line engineering needs to be addressed in a way that effectively supports the development of reusable components. The demand for establishing an explicit link between features and requirements, however, has not been adequately taken into account in software product line methods. This results in a serious problem because components are developed that may not fully meet their requirements.

From one point of view, in order to implement a feature as a highly reusable component, the feature should be at a relatively high level of abstraction. This is reflected in the definition of feature. For example, in FODA [2], Kang defines a feature as: *"a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems"*. On the other hand, to develop a feature into a component that exhibits the expected behavior, requirements related to the feature must be precisely defined and analyzed. Bosch in [5] defines feature as, *"a logical unit of behavior specified by a set of functional and non-functional requirements"*. Therefore, it is important that we establish explicitly links between features and requirements.

Usually, requirements and related concerns of a feature are not fully identified and understood when the feature is defined in feature modeling. Instead, requirements and potential concerns of a feature are only completely explored when it is placed into the context of a concrete software development problem. Therefore, it is essential to establish mappings between a feature model and a problem-oriented software development model, e.g., problem frames, a method for requirements engineering [6]. There have been some attempts made to map features to other models (e.g., UML models). In [7], Laguna et al. define *feature patterns* that can be directly mapped to UML models including class models, use case models, and package models. UML models, however, are solution-oriented. Consequently, mappings between features and UML models do not solve the problem. In this paper, we propose using problem frames with feature models to effectively transform features into reusable components that satisfy product line requirements.

There are reasons for choosing problem frames along with the feature model. First, problem frames can relate a feature to its requirements (problems), domains, and specifications, ensuring that the feature is clearly defined through the requirements. Instead of focusing on the solution, problem frames focus on the problem itself, which needs to be clearly understood before constructing any solution. Moreover, current research [8] [9] [10] shows that a feature model tends to abstract three important concepts- the requirements, domain properties, and specifications. Conversely, in a problem frame, requirements, domain properties, and specifications are the key separate elements. By analyzing a problem based on the distinct concepts of requirement, domain, and specification, the

requirements for the problem are unambiguously derived. Furthermore, problem frames support transformation of a feature into its components. Problem frames are generally recurring problem patterns that have known or pre-defined solutions. Naturally, problem frames allow reusable patterns *(problem and solution patterns)* to be applied in an explicit manner. Therefore, if a feature is realized with a problem frame, known solutions for the development of a component for the feature are available.

Although the problem frame method has been used effectively to derive requirements of a single software system, it has rarely been applied in product line software engineering [11]. The main reason behind this is that a product line domain is far broader and more complicated than that of a single system. This makes it difficult for problem decomposition, a key process in the problem frame method [6]. However, with the introduction of feature modeling, problem frames can be practically and logically applied in product line engineering. A feature model, with its structural knowledge in terms of features, regulates how the concepts in a domain are structured and decomposed. Therefore, a feature model can be used to guide the process of problem decomposition.

The rest of the paper is organized as follows. In section 2, we briefly review the main concepts of the problem frame methodology. In section 3, we outline our proposed method. Sections 4 and 5 discuss the key aspects of the proposed approach. An example is described in section 6, and related researches are discussed in section 7. Finally, section 8 concludes the paper and outlines our future plan.

2 Concepts of Problem Frames

Problem Frames [6], which is a relatively new and emerging method for requirements engineering for single software system development, was first introduced in 1995 by Jackson in the hope that problem frames could capture very complex software problems in a simple way by systematically structuring or decomposing the problem into sub-problems. There are two main concepts behind the problem frame methodology: *problem decomposition* and *problem recognition*. Problem decomposition is that a large problem should be decomposed into relatively small sub-problems to handle its complexity. Through problem decomposition, we hope to factor out the sub-problems that are recurring problem patterns. Problem recognition implies that a problem has its own characteristics and should be recognized as a certain problem class which has known solutions. The problem, therefore, will be solved by reliable solutions.

A problem frame defines an intuitively identifiable problem class in terms of its context which is encoded as domain, specification, and requirement. A problem frame is represented using the problem frame diagram which consists of a problem domain or a physical world (W), requirements (R), and machine (S) (Fig. 1). The problem domain is denoted by a plain rectangle describing the given properties of the problem world or the problem surroundings. The requirements, represented by a dotted oval, describe the certain effects or required phenomena (a) that we wish to be true in the problem domain. The dotted line represents

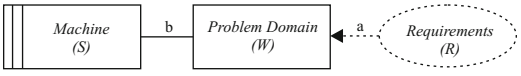


Fig. 1. Problem frame diagram

the requirements in reference to phenomena, which are events, states, entities, and/or values, that we can observe in the problem domain. The machine, which is denoted by a rectangle with a double vertical stripe, describes, in terms of specifications, the behavior (b) of the machine that must be exhibited in the information system to realize the requirements. Note that a rectangle with a single vertical line is used instead of a plain one to indicate that the problem domain is a design domain which needs to be further refined (Fig. 5e). A correctness argument then makes sure that the relationship ($\vdash$) between S, W and R is satisfied. It means that given the domain properties W and the machine that we want to build with specifications S, the requirements R must be satisfied.

To date, Jackson has introduced six important problem frames: *Required Behavior*, *Commanded Behavior*, *Information Display*, *Commanded Information Display*, *Transformation*, and *Workpieces*, each of which represents a recurring problem pattern at a high level of abstraction so that it is concise, easily understood, and applicable to many domains in reality. The required behavior problem frame, for example, is described by Jackson [6] as follows: *"There is some part of the physical world whose behavior is to be controlled so that it satisfies certain conditions. The problem is to build a machine that will impose that control."* The description implies that whenever a software problem manifests a core concern of required behavior, it can be captured and analyzed with a required behavior problem frame. A problem frame, therefore, represents a certain core concern which can have many variants and manifestations in different forms. Obviously, the core concerns are captured in the problem frame model. Natural questions that arise are: Are the "concerns" represented in a feature model? How do we recognize the concerns? The answer to the first question is apparently *"yes"* since both features and problem frames are abstraction of software requirements which contain concerns. This means that mappings between a feature model and a problem frame model exist. If we answer the later question, the product line engineering area could benefit from it: the concerns implicated by a feature could then be identified early, analyzed with the problem frame method, and finally solved with known and reliable solutions. Based on the application of problem frames and feature modeling, the reuse paradigm can be applied extensively linking domain problems to domain solutions. The next section will outline how this idea is implemented.

3 Outline of the Method

The main idea of the proposed method is to apply the concept (i.e., requirements, domains, and specifications) of problem frames to SPLE. As a starting point, a feature model, which is created in domain engineering, abstracts a software product line in terms of features. Next, the feature model is mapped to

problem frames by applying the mapping heuristics derived from feature modeling and a set of feature mapping units (to be discussed in section 4). Through this mapping, each feature is realized with a problem frame from which the feature's requirements and potential concerns are clearly identified and analyzed by applying problem analysis. Finally, the problem frames are transformed to components using the UML-based approach for problem frame oriented software development [12]. Here, we do not aim to develop a complete product line method; instead, we focus only on the domain engineering in which reusable artifacts (e.g., components) are developed.

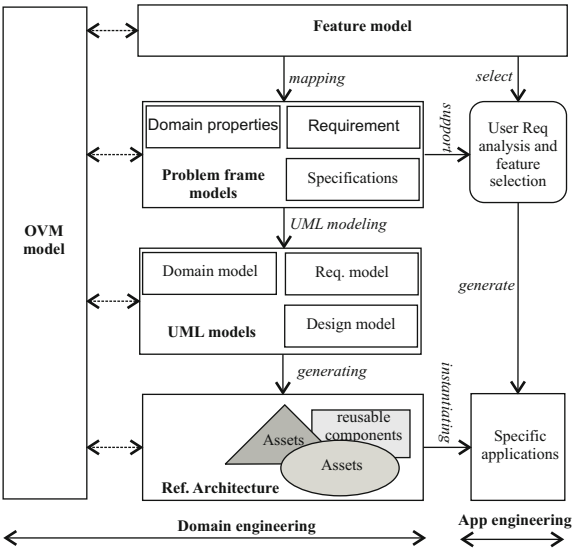


Fig. 2. Outline of the proposed method

The steps for relating features to reusable components are summarized as follows (Fig. 2):

1. Represent the problem space of a product line with a feature model.
2. Develop mappings between features and problem frames by applying mapping heuristics and feature mapping units.
3. Model solutions: model components for features using known solutions for their corresponding problem frames (using UML).

The orthogonal variability model (OVM) may be used to keep the variability consistent across the models.

4 Mapping between Feature Models and Problem Frames

From our experience of applying the approach in domains including Home Integration System (HIS) and Elevator product lines, we understand that one of

the most important factors of the approach is the mappings between the problem frames and features; that is, how we precisely realize features with problem frames. This eventually depends on the organization or structure of a feature model. There are many variants of feature model proposed by various product line methods [3]. However, it is our experience that the feature model proposed by FORM can support our approach. The following section explains how the mapping heuristics are derived from the FORM feature model.

4.1 Mapping Heuristics

In FORM, the idea of classifying features into four layers, called *Capability (CA)*, *Operating Environment (OE)*, *Domain Technology (DT)*, and *Implementation Technique (IT)*, is similar to *requirements*, *domains*, and *specifications* in problem frames. Features in the CA layer are at the highest level of requirement abstraction that is understandable to both users and developers. In the problem frame framework, requirements are modeled with problem frames (or problems which need to be further decomposed into recognized problem frames). Consequently, features classified as Capability correspond to *requirements R* of problem frames or problems. Features in the OE layer are close to the domain properties (*physical world W*) of problem frames in the sense that they both describe the environment of a system or systems. Therefore, those features are mapped to the domain properties of a problem frame. The DT layer includes features that describe solutions pertaining to a specific domain, thus these features correspond to *specifications S* or the *machine* of problem frames. Likewise, features belonging to the IT layer represent solutions at a lower level of abstraction compared with that of the DT layer, and they also correspond to *specifications S* of problem frames. This observation allows us to derive the first heuristic as follows. ***Heuristic 1.*** *Let f(x) denote that a feature x belongs to a layer f, then: - CA(x) is mapped to a problem (represented by R), or a problem frame (represented by machine S); - OE(x) is mapped to a domain W; - DT(x) or IT(x) is mapped to a specification S.*

Features are organized into layers using the relationships, *composed-of*, *generalization*, and *implemented-by*, such that the organization of a feature model represents the structure of a domain. A feature is structurally related to its sub-features by the *composed-of* relationship, a representation of an aggregation defined as: "*the abstraction of a collection of units (e.g., features) into a new unit (e.g., a feature)*". "*Refining an aggregation into its constituent units is called decomposition*" [2]. This is similar to problem decomposition in the problem frame methodology, in which a whole problem is decomposed into sub-problems so that the decomposition should represent the best natural structure of the problem (domain). Consequently, *composed-of* relationships in a feature model implicate how a problem in a corresponding problem frame model should be decomposed into sub-problems: this results in the following heuristic: ***Heuristic 2.*** *The composed-of relationship guides the process of problem decomposition.*

When features f and g are generalized into a feature h, it means that the commonalities between f and g are abstracted into a new feature h. When features

are organized with the generalization relationship, they must belong to the same layer. This observation lets us define the third heuristic: ***Heuristic 3.*** *If features are organized with the generalization relationship, their corresponding mappings into problem frames must be made to the same type of problem frame element (i.e., requirements, domain, or specification).*

If a feature f is implemented by a feature g, then g is a known solution of f, and g belongs to either layers DT or IT. Therefore, when it is mapped to a problem frame model, g will become a specification S, i.e., a solution of f. This means that we can safely skip the mappings of the features that are organized by the *implemented-by* relationship as they have defined solutions. Thus we have the following heuristic: ***Heuristic 4.*** *Features that are organized by the implemented-by relationship can be safely removed from a feature model before mapping it to problem frames.*

These heuristics imply a strong relationship between the two models. This connection manifests itself in two ways. First, there are mappings between features of a feature model and artifacts (requirements, domains, and specifications) of a problem frame. Second, there is similarity between the structural relationships between features in a feature model and the structural decompositions in problem frames. A question that appears here is how we can precisely realize a feature or features with a problem frame(s) (e.g., a required behavior).

The next section will describe how a group of features can be systematically mapped to a certain frame in terms of feature mapping units.

4.2 Feature Mapping Units

Realization of a feature or features with a problem frame is made possible using *feature mapping units (FMU)* of four types of core concerns, i.e., required behavior, commanded behavior, information display, and inforamtion transformation. A FMU has its root feature in the CA layer. Sub-features are related to their root via *composed-of* relationships. Since a FMU is mapped to a problem frame, the *behaviors* implied by the FMU type must be exhibited by its member features. The four FMU types are defined below.

Required Behavior FMU. If a feature f_1 (requiring feature) requires feature f_2 (required feature) to represent its behaviors α, β, γ at a particular time during f_1's operation, then feature f_1 and f_2 form a required behavior FMU.

- Mappings: f_1 will be mapped to the machine domain *(S)*, f_2 and its behaviors α, β, γ will be mapped to the controlled domain (W[α, β, and γ]) that is required to represent behaviors α, β, γ at the time t_α, t_β, t_γ respectively, to satisfy the *required behavior* requirements *(R)* (Fig. 3a).

- Example: The *fire action* and *sprinkler* features together form a required behavior FMU because the *fire action* feature needs the *sprinkler* feature to either activate it when a fire event is detected or else to deactivate.

Commanded Behavior FMU. If the behaviors of a feature (f_1) are controlled by a feature (f_2) that functions in accordance with the commands issued by a user, then features f_1 and f_2 form a commanded behavior FMU.

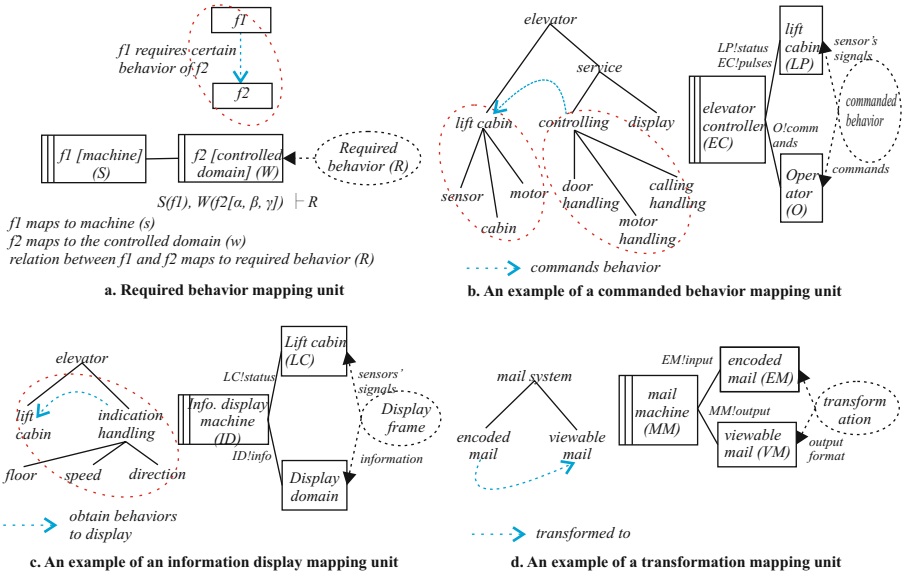


Fig. 3. Feature mapping units

- Mappings: f_2 will be mapped to the machine domain *(S)*, and f_1 with its behaviors α, β, γ will be mapped to the controlled domain (W[α, β, and γ]) that is commanded to represent behaviors α, β, γ when the user issues $command_\alpha$, $command_\beta$, $command_\gamma$ respectively, to satisfy the *commanded behavior* requirements *(R)*.

- Example: The *controlling* feature and the *lift cabin* feature form a commanded behavior FMU because the former commands the behaviors (e.g., stopped, moving) of the latter whenever an operator of the elevator issues a command(e.g., a floor register) (Fig. 3b).

Information Display FMU. When a feature (f_1) sends a request to display the behaviors or states of other feature (f_2), we define f_1 and f_2 form an information display FMU.

- Mappings: f_1 will be mapped to the machine domain *(S)*, f_2 and its behaviors α, β, γ will be mapped to the controlled domain (W[α, β, γ]) whose behaviors or states α, β, γ need to be displayed on the display domain.

- Example: The *lift cabin* feature and the *indication handling* feature form an information display FMU because the latter seeks to display behaviors and states *(e.g., current floor, direction, speed)* of the former (Fig. 3c).

Transformation FMU. When one of the two features (f_1) plays the role of feeding the input to the other feature (f_2), with computer-readable information, to output it in a particular format, we say f_1 and f_2 form a transformation FMU.

- Mappings: f_1 will be mapped to the input domain *(W[input])*, and f_2 will be mapped to the output domain *(W[output])*.

- Example: The *encoded mail* feature and the *viewable mail* feature form a transformation FMU because the encoded mail must be transformed to a viewable format (Fig. 3d).

5 Solution Modeling

5.1 Problem Analysis

Once a feature model is mapped to problem frame models, problem analysis is conducted to help developers clearly understand each feature and realize the potential concerns by analyzing the corresponding problem frames of the feature. This analysis is based on a correctness argument, which brings together a problem frame's requirements, domain properties, and machine specifications to discharge the problem frame's concerns and ultimately make sure that the requirements are satisfied. Because a problem frame defines a class of problems which have common and relevant characteristics, it has recurring concerns. For example, by analyzing a required behavior problem frame which corresponds to features *(fire action, locker, sprinkler, and alarm)* (Fig. 5b-f), we can identify concerns that may cause strange behaviors resulting in dissatisfaction of the requirements. One example would be concerns related to behaviors of problem domains *(sprinkler and alarm)* where the *alarm* is turned on but the *sprinkler* does not work because its *water source* is empty. Another example is the concerns related to machine specifications, for instance, the control signals that activate the *sprinkler* but turn off the *alarm*.

The analysis of the concerns essentially shapes architectural components implementing the problem frames. Many research examples [13] [14] show the explicit links between problem frames and architectural styles. Therefore, in our approach we apply the research results to select architectural styles that are best suited for the development of a component during the analysis. For example, when a feature f is realized with a required behavior frame, the architectural style for a component implementing the feature f would be the *Layered style*. The output of this analysis is organized in a tabular format. Row i represents a FMU f_i and its related analysis factors- Problem frames (PF_i and requirement R_i) corresponding to feature f_i, Phenomena (P_i) observed on PF_i, Domains (D_i) within PF_i, Concerns (C_i) related to PF_i, Architectural styles (AS_i) recommended for PF_i. A problem analysis for the *Fire action* feature (Fig. 4) was conducted and its results are shown in the Table 1.

After this step, major concerns as well as potential issues of a feature can be discovered and analyzed. Based on this analysis, possible architectural styles for components that implement features are introduced. The following section will describe the final step involving transforming the problem frames to reusable components.

Table 1. A sample of analysis output: from features to architectural styles

FMU	Problem frames	Phenomena	Domains	Concerns	A. Style
f_1: {*Fire action, ... sprinkler, and alarm*}	PF_1: Required behavior	P_1: -Events: acting, detecting; -States: activated, detected; -Interface: status signals, controlling signals	D_1: Sprinkler, alarm, environment	C_1: Changing behavior in real-time; Hardware concerns	AS_1: Layered style
f_2: {*Fire detection, smoke sensor, and moisture sensor*}	PF_2: Transformation	P_2: -Events: fire event; -Status: working/not working; -interface: Status signal, raw signal, output signal	D_2: sensors, environment	C_2: Reliability of sensor signals because of bad environment conditions; sensor malfunction	AS_2: Layered style or pile and filter

5.2 Component Modeling

Problem Frames to Reusable Components. Our goal is to generate specifications for components that implement features using a modeling language. In our approach, we use the UML-based method in [12] to model the problem frames. There are some reasons for choosing this method. First, developers are familiar with UML because its concepts and notations are highly standardized and widely used. Second, a problem frame is defined in terms of three separate artifacts of requirements, domain properties, and specifications; and the UML-based method provides ways to specify these [12]. The *requirements* are modeled using class diagrams and use case diagrams; the *domains* are modeled by class diagrams, behavior diagrams, and sequence diagrams; the *specifications* are modeled using class diagrams, object diagrams, behavior diagrams, and package diagrams.

To map a simple problem frame to a component, we propose a general scheme in which the machine domain and problem domains of a problem frame are modeled with classes using the *object-oriented* modeling method. The machine domain is modeled with a *MachineDomain* class. The problem domains are modeled with *ControlledDomain* classes. Interactions between the machine domain and problem domains are modeled with interface classes that are realized or implemented by the *MachineDomain* and *ControlledDomain* classes.

Variability Management. Although the UML-based approach can be used for modeling problem frames, it does not mention how to deal with design variations. One of the major difficulties in product line engineering is how to manage variations that are captured by a feature model. When features are mapped to problem frames, variations in a feature model must also be consistently mapped to the problem frame model. Variations captured by the feature model must be reflected as variations in requirements, domain properties, or specifications of

the problem frame model. Variations in the problem frame model must finally be reflected to the architectural component model.

However, problem frames and UML are not equipped with the concept of intended variability. In our method, we employ the orthogonal variability management (OVM) [15]. OVM manifests itself as a flexible and practical model for managing variability. Because it is an orthogonal model, it can be independently applied for any model in which variability exists. Using OVM, all the variation points in a feature model will be consistently referenced to those in its corresponding problem frame models and UML models. For example, there is a variation point *detection* in which the *heat detection* feature that detects *fire* with a *thermal sensor (heat sensor)* is an optional feature (Fig. 4). The variation point *detection*, with its variant *heat sensor*, is explicitly mapped to the problem frame model that has an optional domain *heat sensor* and to the UML model which states that the *HeatSensor* object is optional (Fig. 6).

6 Example: Home Integration System (HIS)

6.1 Introduction to HIS

HIS is a home integration system product line which controls and manages a collection of devices to maintain the security and safety of a building or a house. The main features of HIS for illustrating the proposed approach are described below and organized using a feature model *(after applying the heuristic 4 to simplify the feature model).*

Fire Detection and Action: The fire events are detected by smoke detectors and heat sensors installed in the house. When a fire event is detected, HIS turns the alarm and all the sprinklers on and unlocks all HIS-controlled doors. Once the fire is under control, the alarm and all the sprinklers will be turned off but the doors will remain unlocked for the duration of the time preset by the owner.

Intrusion Detection and Action: Intrusion events are detected by the motion sensors. When an intrusion event is detected, HIS turns on the alarm and locks all the HIS-controlled doors.

Flood Detection and Action: Flood events are detected by the moisture sensors. When a flood event is detected, HIS shuts off the water main to the house. When moisture is detected on the basement floor, the sump pump will be activated.

6.2 From Feature Models to Problem Frame Models

First, the root feature *HIS*, which is comprised of three features: *Intrusion*, *Fire*, and *Flood*, is mapped to the *HIS problem* which is then decomposed into three corresponding sub-problems, *P_Intrusion*, *P_Fire*, and *P_Flood (heuristic 2)*. The decomposition of these problems can take place in parallel since they are independent from each other. Here, the process of further decomposing the

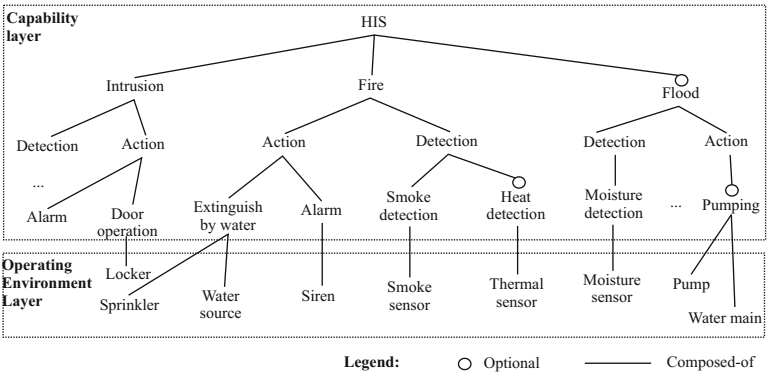


Fig. 4. The feature model of HIS

P_Fire problem is described in detail. According to the HIS feature model, the *Fire* feature is comprised of two features, *Detection* and *Action*. Consequently, by applying heuristic 2, the *P_Fire* problem will be further decomposed into two corresponding sub-problems, *P_Detection* and *P_Action* (Fig. 5a-d-e).

P_Action, A Required Behavior Frame. From feature modeling, we know that the *door operation* feature requires the behaviors of the *locker* feature be *open* or *locked*, therefore the two features form a required behavior FMU. Likewise, the {*water, sprinkler*} features and the {*alarm, siren*} features are required behavior FMUs. Therefore, the *action* feature which is comprised of three FMUs is realized by a required behavior problem frame (Fig. 5b-f).

P_Detection, A Transformation Frame. The *smoke sensor* and *heat sensor* features feed raw signals on the heat and smoke information of the surroundings to the *detection* feature which then decides whether there is a fire event or not. Therefore, the *detection*, *smoke sensor*, and *heat sensor* features form a transformation FMU which is realized by a transformation problem frame (Fig. 5c-g).

6.3 From Problem Frames to Reusable Components

We will move on to demonstrate the development of the *C_Detection* and *C_Action* architectural components from their corresponding problem frames, *P_Detection* and *P_Action*. The *fire detection* and *fire action* features are realized by the *P_Detection* and *P_Action* problem frames which represent a recognized problem in terms of requirements, domains, and specifications. First, problem analysis is conducted to determine a correctness argument which ensures that the *R_Detection* and *R_Action* requirements are satisfied (Fig. 6). Many of the potential concerns and issues which may affect the satisfaction of the requirement are explored. Also, architectural styles best for developing the components are identified. A sample of the problem analysis for *P_Detection* and *P_Action* problems

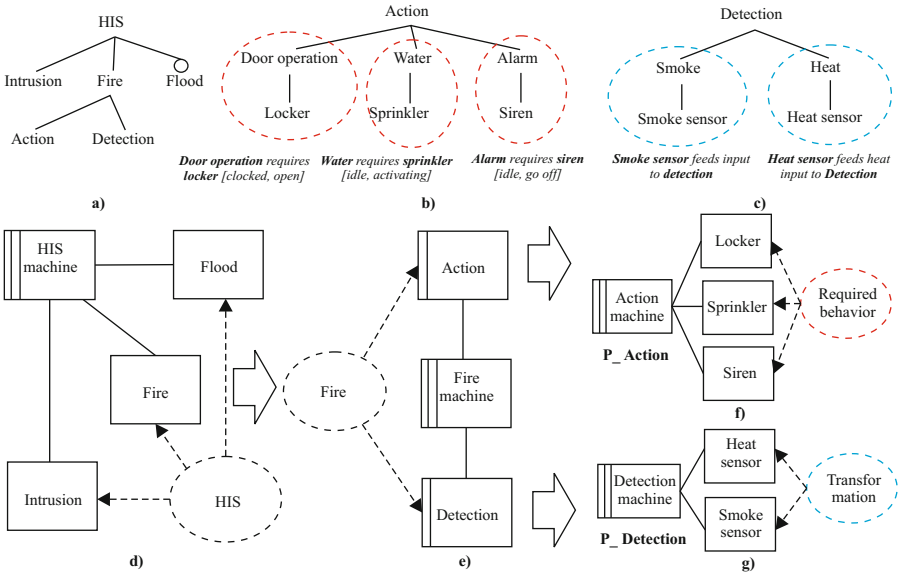


Fig. 5. From feature models to problem frames

is documented in the Table 1. Next, the architectural components, *C_Detection* and *C_Action*, are modeled using the UML specification language. Interactions and communication between the artifacts of a problem frame (machine domains and controlled domains) are critical for understanding and mapping the frame to the UML component model.

C_Detection Component. The problem domain, which includes the *Heat sensor* and *Smoke sensor* sub-domains of the transformation problem frame, is modeled with the *DetectionDomain* class which is comprised of the *HeatSensor* and *SmokeSensor* classes of the sub-domains (Fig. 6). The interactions *(a, b, c)* between the problem domain and the machine are modeled by the interfaces: - *SensorSignal*, which sends data on *heat* and *smoke* to the machine; - *FireDetectionInfo*, which abstracts detection information about *detected fires* and *non-detected fires* issued from the machine. The *DetectionDomain* class then implements the *SensorSignal* interface. The machine domain is modeled with the *DetectionMachine* class which realizes the *SensorSignal* interface and implements the *FireDetectionInfo* interface.

C_Action Component. The problem domain, which includes the *Locker*, *Sprinkler*, and *Siren* sub-domains of the required behavior problem frame, is modeled with the *ActionDomain* class. The *ActionDomain* class is comprised of the *Locker*, *Sprinkler*, and *Siren* classes of these corresponding sub-domains (Fig. 6). The interactions *(d, e, f)* between the problem domain and the machine are modeled by the interfaces: - *StatusSignal*, which communicates the status of

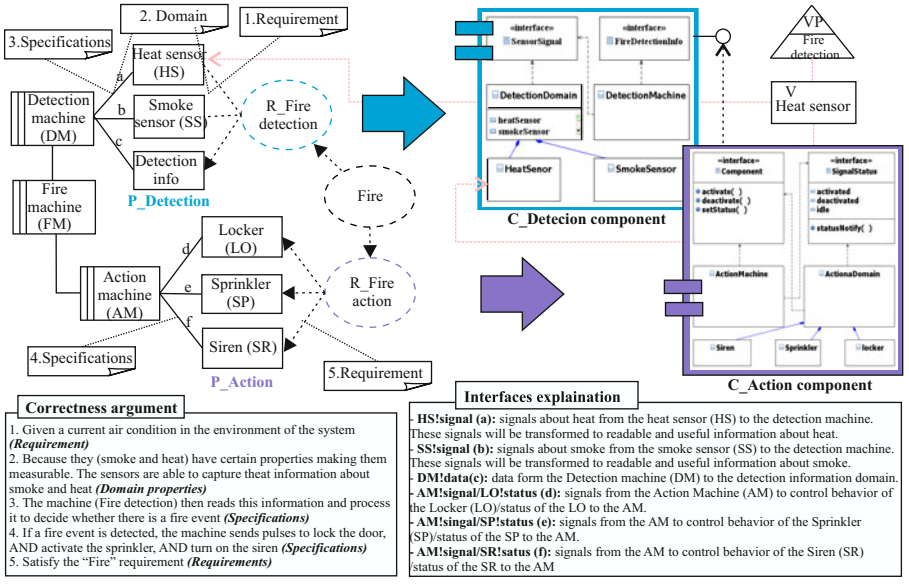


Fig. 6. Mapping problem frames to C_detection and C_action components

the *Locker*, *Sprinkler*, and *Siren* domains to the machine; - *Command*, which abstracts the commands of *activate* and *deactivate* issued by the machine. The *ActionDomain* class then implements the *StatusSignal* interface and realizes the *Command* interface. The machine domain is modeled with the *ActionMachine* class which realizes the *StatusSignal* interface and implements the *Command* interface.

7 Related Works

There has not been much work done to try to integrate problem frames into product line software engineering. Approaches that use problem frames to compensate feature models mainly focus on managing variability. In [10], Salifu et al. suggested using problem descriptions to represent and analyze variability in context-aware software products. Changes in the properties of external domains are captured and analyzed to see how they affect the products' behavior. This approach is good at describing unintentional variability in the product line context (domains). However it did not consider representing the intended variability of requirements, and it is difficult to apply in product line engineering.

Classen et al. [8], proposed problem frames instead of feature diagrams to analyze the variability in product families. They claimed that feature diagrams tend to mix up three important descriptions: requirements, domain properties, and specifications. Therefore they are inadequate for capturing problem structures. However, the relationships between a feature model and a problem frame

model are not mentioned. Without a feature model, representing and analyzing variability in a large domain with problem frames is complicated and impractical.

Zou [11] introduced extended notations to support requirements, domains, and machine variability in problem frames so that problem frames can be applied to product line engineering. However, these additional notations make the problem frames diagrams more complicated and may change their original meaning. It is not practical to model variability using problem frames, especially when the target domain is complex.

Classen et al. [9] claimed that concepts in each feature should be requirements, domain properties, and specifications. This idea was applied to identify and solve interactions between features (i.e., feature interaction problems) in a product line domain.

Even though the researches mentioned above applied problem frames to product line engineering, they inadequately addressed the relationship between the feature model and problem frames (features and requirements).

8 Conclusion

In this paper, we introduced an approach in which feature models and problem frames are integrated in an attempt to effectively develop reusable components in product line software engineering. The central idea of this approach is the application of the concepts of problem frames and feature modeling to realize the *feature mapping units* from which reusable components are modeled and developed. The most important benefit of this approach is that the reuse paradigm is extensively applied in both the problem space and the solution space of a product line. For future research, we plan to apply this method in more practical and larger domains with complex feature models. This will mean that the approach will be thoroughly validated. We also plan to further enhance and formalize the method. To date, mappings between feature models and problem frames have been made through feature mapping units and using mapping heuristics rather than a formal approach. The last challenging future work option is to develop a tool that supports this approach.

References

1. Kang, K.C., Lee, J., Donohoe, P.: Feature-oriented product line engineering. IEEE Software 19, 58–65 (2002)
2. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute (November 1990)
3. Robak, S.: Feature modeling notations for system families. In: ICSE 2003 Workshop on Software Variability Management, Portland, Oregon, pp. 58–62 (2003)
4. Kang, K.C., Kim, S., Lee, J., Kim, K., Kim, G.J., Shin, E.: Form: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering 5, 143–168 (1998)

5. Bosch, J.: Design and use of software architectures: adopting and evolving a product-line approach. ACM Press/Addison-Wesley Publishing Co., New York (2000)
6. Jackson: Problem Frames: Analysis and Structuring Software Development Problem. Addison-Wesley Professional, MA (2001)
7. Laguna, M.A., González-Baixauli, B., Marqués Corral, J.M.: Feature patterns and multi-paradigm variability models. Technical Report 2008/01, Grupo GIRO, Departamento de Informática (May 2008)
8. Classen, A., Heymans, P., Laney, R., Nuseibeh, B., Tun, T.T.: On the structure of problem variability: From feature diagrams to problem frames. In: Pohl, K., Heymans, P., Kang, K.C., Metzger, A. (eds.) Proceedings of the First International Workshop on Variability Modelling of Software-intensive Systems, Limerick, Ireland, LERO, pp. 109–117 (January 2007)
9. Classen, A., Heymans, P., Schobbens, P.Y.: What's in a feature: A requirements engineering perspective. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 16–30. Springer, Heidelberg (2008)
10. Salifu, M., Nuseibeh, B., Rapanotti, L., Tun, T.T.: Using problem descriptions to represent variabilities for context-aware applications. In: VaMoS, pp. 149–156 (2007)
11. Zuo, H., Mannion, M., Sellier, D., Foley, R.: An extension of problem frame notation for software product lines. In: APSEC 2005: Proceedings of the 12th Asia-Pacific Software Engineering Conference, Washington, DC, USA, pp. 499–505. IEEE Computer Society, Los Alamitos (2005)
12. Choppy, C., Reggio, G.: A uml-based approach for problem frame oriented software development. Information and Software Technology 47(14), 929–954 (2005); Special Issue on Problem Frames
13. Choppy, C., Hatebur, D., Heisel, M.: Architectural patterns for problem frames. Software, IEE Proceedings 152(4), 198–208 (2005)
14. Wang, C., Depei, Q., Chuda, L.: Architecture-based problem frames constructing for software reuse. In: IWAAPF 2006: Proceedings of the 2006 international workshop on Advances and applications of problem frames, pp. 19–24. ACM, New York (2006)
15. Pohl, K., Metzger, A.: Variability management in software product line engineering. In: ICSE 2006: Proceedings of the 28th international conference on Software engineering, pp. 1049–1050. ACM, New York (2006)

Eliciting and Capturing Business Goals to Inform a Product Line's Business Case and Architecture

Paul Clements[1], John D. McGregor[2], and Len Bass[1]

[1] Software Engineering Institute, Carnegie Mellon University,
4500 Fifth Avenue, Pittsburgh PA 15213 USA
[2] Department of Computer Science, Clemson University, Clemson SC 29634 USA
clements@sei.cmu.edu, johnmc@cs.clemson.edu, lenbass@cmu.edu

Abstract. Business goals constitute an important kind of knowledge for a software product line. They inform the product line's business case and they inform its architecture and quality attribute requirements. This paper establishes the connection between business goals and a product line's business case and architecture. It then presents a set of common business goal categories, gleaned from a systematic search of the business literature that can be used to elicit an organization's business goals from key stakeholders. Finally, it presents a well-defined method, which we have tried out in practice, for eliciting and capturing business goals and tying them to quality attribute requirements.

Keywords: Software product line, architecture, product line architecture, business case, business goals, business goal scenario, quality attribute requirements.

1 Introduction

Developing similar software systems together as a software product line can bring about improvements – sometimes of an order of magnitude – in cost, time to delivery, and quality. Software product lines represent a strategy that results in a sizable investment over its lifetime. Such an investment requires a justification, often in the form of a *business case*. Once the investment is made, a *product line architecture* serves as the backbone of the technical effort.

Both of these critical product line artifacts (the business case and the architecture) are well-served by an explicit articulation of the development and customer organizations' business goals. Business goals describe the objectives an organization wishes to accomplish. A business case can only help determine if a product will serve an organization's interests if those interests are known and captured. A product line architecture must deliver necessary quality attributes to the product line at large as well as individual products and, as we will show, business goals have a direct and fundamental influence on those quality attributes.

However, business goals are not always written down in an accessible fashion; many times they're implicit and have never been identified.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 393–405, 2010.

This paper offers three main contributions.

- First, it shows the connection between business goals and a product line's business case and architecture. In particular, it shows how business goals can lead to specific quality attribute requirements for a system or a family (product line) of systems. This relationship is critical for the architect (or product line architect) to understand and use, but until now has remained implicit at best and ignored at worst.
- Second, it presents a set of common business goal categories, gleaned from a systematic search of the business literature that can be used to elicit an organization's business goals from key stakeholders. These business goals can inform the product line's business case and its architecture.
- Third, it presents a well-defined method, which we have tried out in practice, for eliciting and capturing business goals and tying them to quality attribute requirements.

2 Business Goals and the Product Line Business Case

A business case is, briefly, a justification of an action:

> *A business case is a tool that helps you make business decisions by predicting how they will affect your organization. Initially, the decision will be a go/no-go for pursuing a new business opportunity or approach. After initiation, the business case is reviewed to assess the accuracy of initial estimates and then updated to examine new or alternative angles on the opportunity... In this role, management uses the business case to determine possible courses of action.[2]*

A primary purpose of a business case is showing how a candidate strategy will help the organization meet its business goals. Strategies that can be examined in the light of a business case include whether to build a set of products as a product line or individually; whether to add a new product to the product line or build it separately; whether to expand the scope of the product line (requiring the core assets to be modified), and others. One way to view the business case is that it allows management to ask "Is this strategy we are considering consistent with our business goals? Will it help us achieve them?" In order for that to happen, those business goals have to be known and articulated.

A product line business case goes hand-in-hand with the product line scope (which defines what products are in the product line and which are not). A product line with a scope that is too narrow will not provide sufficient return on investment. A product line with a scope that is too broad could collapse under its own weight, due to the costs of being overly general. This close, mutually informative relationship between business case and scope is captured in the "What to Build" product line pattern [6]. The pattern, shown in Figure 1, captures the relationship of the business case with other practices. It shows that creating the business case is informed by information from the Understanding Relevant Domains, Market Analysis and the Technology Forecasting practices.

- Understanding Relevant Domains – The business case reflects the culture of the organizations building the products. The culture of an aircraft manufacturer

is different from that of a financial institution. The factors considered in their business cases will be quite different.

- Market Analysis - The data from the market analysis provides a basis for projecting sales and income for the products being justified.
- Technology Forecasting – The technology forecast predicts the need for new technologies that will cost to acquire and that may require training for personnel.

In addition to technology, the social and legal environments may also change during the lifetime of the product line and these changes should also be forecast [14]..

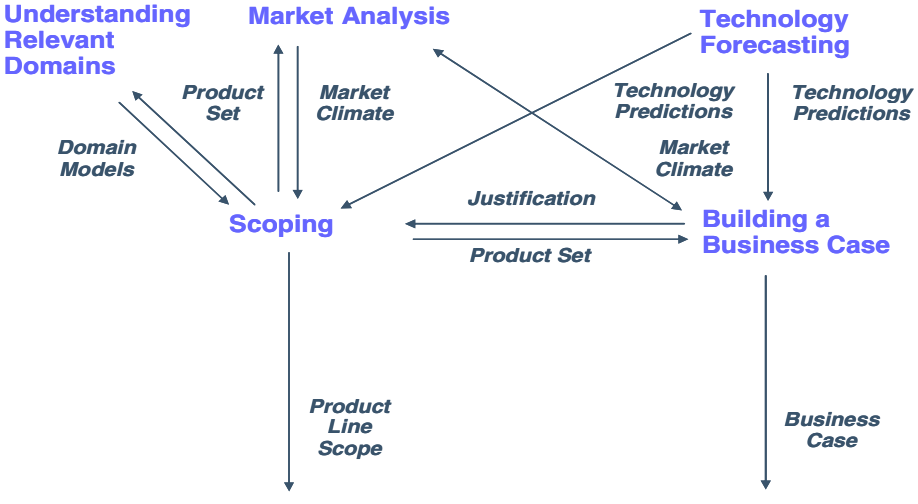


Fig. 1. The "What to Build" pattern [6]. The arrows mean "informs."

As with many things, a business case is something that you don't hear about when it works well, but a poor or missing business case can derail the entire product line effort. One of the most serious flaws in a business case is failing to take into account the interests and motivations of all of the important stakeholders, who can be the "owners" or instigators of an organization's business goals.

The business case aids an executive in making an investment decision by providing answers to critical questions. Questions that a comprehensive business case should answer include the following [1]:

1. What problem are we trying to solve?
2. What needs to change, how much, by when?
3. What are possible solutions to meet the need?
4. What are the two or three likely solutions?
5. What are the costs, benefits, and risks of each viable alternative?
6. What are the return-on-investment metrics?
7. Which is the "best" alternate solution?

In a list such as this, having the business goals explicitly articulated can help answer whether and how an alternative under consideration would be beneficial.

Business case practices for a product line effort differ from those for a single system only in the nature of the changes being considered and analyzed. The organization is making an economic case built on the current costs of doing business and the benefits derived from following the strategic direction. The initial go/no-go decision answers the question: "Do we build the set of products we're considering as a product line or not?"

Cohen lays out steps for building a business case for a product line [8]:

1. Establish Product Line Context
2. Estimate the Likely Costs and Analyze Potential Risks for All Alternatives
3. Estimate the Likely Benefits and Contrast With Current Business Practice
4. Develop a Proposal for Proceeding
5. Close the Deal: Make Final Adjustments and Proceed to Development

Once again, business goals can be seen as informing this process. They form an indispensable part of the context, and actually establish precisely what constitutes "benefit."

Many companies have their own outlines for a business case but we have one that we have used with several customers. It includes the following sections:

1. Mapping of the business goals to the product line strategy
2. Description of alternative approaches to meeting the goals
3. Analysis of the strengths/weaknesses/ opportunities/threats associated with the product line (SWOT analysis)
4. Analysis of strategic factors that influence the product line strategy
5. Comprehensive scenario analysis including cost/benefits
6. Recommended course of action

Sections 1 and 3 of the outline describe the strategic direction of the product line. This is the business justification for using the product line approach. We have used Porter's Five Forces model for strategy development when developing the production strategy with customers [15]. The forces in this model drive the identification of business goals related to balancing those external forces with internal actions.

Section 5 of the business case outline obviously has a direct relationship with business goals, which in effect define what constitutes a "cost" and a "benefit." While these could have strictly monetary implications, other interpretations are certainly possible. For example, a business goal having to do with meeting responsibilities to employees could reveal as beneficial a strategy that helps developers go home on the weekends.

Equation 1 shows the basic formula used in the product line cost modeling language SIMPLE [7] to compute a quantitative justification. Some information needed to evaluate that formula is not directly financial. The business goals for a product line inform the analysis of a candidate strategy by defining such parameters as the timeline for product deliveries (which is needed for the value of "*t*" in the SIMPLE formula), anticipated revenue streams that result from by-products of the core asset development activities included as one of the benefit (*Ben*) functions, and anticipated refresh

cycles of the core assets based on obsolescence, which cuts across each cost function. These values affect the content of the SIMPLE formula.

Equation 1 SIMPLE Formula

$$\sum_{j=1}^{m} Ben_j - (C_{org}(t) + C_{cab}(t) + \sum_{i=1}^{n} (C_{unique}(product_i, t) + C_{reuse}(product_i, t)))$$

This equation provides clues to some business goals that may be implicit and should be made explicit during elicitation.

- Ben is a function that captures quantitative benefits of a strategy. This term reminds us about the need to elicit goals related to revenue streams and non-tangible benefits such as increased customer confidence or loyalty that may be quantifiable. Non-tangible benefits are not only on the customer side but may, as we will see, occur on the developing organization side as well.
- C_{org} is a function capturing the organizational cost of a strategy. To calculate C_{org}, elicitation should explore whether there are any business goals related to evolving or resizing the organization. This will impact the cost of running the organization.
- $C_{cab}$is the cost of establishing the core asset base. If there are goals to sell the core asset base this will require extra costs for bundling, security, licensing, etc.; there may be major quality improvement initiatives that will affect the cost of the core assets
- C_{unique} is the cost of any development that is unique to a particular product. Here there may be goals of improving the market position of the organization by mining unique code to produce new core assets
- C_{reuse} is the cost of reusing core assets to build a product. Process changes may affect the cost of reusing assets by increasing the cost in the short run but decreasing it in the long run

Regardless of the questions, steps, or outline used to capture a business case, there are some cross-cutting issues that should be made explicit.

1. First, the time horizon for the business case for a software product line is different from the business case for a single product. This justification covers multiple products and the operation of an on-going production capability. Estimates for products that will be built further into the future probably will not be as accurate as those for a product or products that will be built immediately. One reason for this is that business goals can and often do change over time [14], and any method to capture them must take that into account.
2. Second, the core assets complicate the computation of costs and benefits. Assets are created and maintained on an on-going basis. A number of product line organizations such as Nokia and Overwatch Textron supplemented the benefits derived from their product line by selling the core assets to other product line organizations [10]. Having the assets available clearly has value for the organization, but there are no clear guidelines for how to balance that value against the development and maintenance costs.

3 Business Goals and the Product Line Architecture

Here's what the SEI's Framework for Software Product Line Practice [2] has to say about the product line architecture:

> *The product line architecture is an early and prominent member in the collection of core assets. The architecture is expected to persist over the life of the product line and to change relatively little and slowly over time. The architecture defines the set of software components (and hence their supporting assets such as documentation and test artifacts) that populates the core asset base. The product line architecture–together with the production plan–provides the prescription… for how products are built from core assets. Once it's been placed in the core asset base for the product line, the architecture is used to create product architectures for each new product.*

Like all software architectures, one of a product line architecture's primary obligations is to deliver the necessary quality attributes to the individual products. It has been argued elsewhere [4] that achievement of quality attributes is in fact the architecture's most compelling reason for existence. In a product line, quality attribute achievement is even more important, because the various products in the product line may exhibit different quality attributes but the same architecture should apply to all.

Properties of a system that help determine whether it satisfies the fitness criteria derive from business goals. Otherwise, why do they exist? If we ask, for example, "*Why* do you want this system to have a really fast response time?" we might hear that this will differentiate the product from its competition and let the developing organization capture market share, or that this will make the end user more effective and this is the mission of the acquiring organization, or other reasons having to do with the satisfaction of some business goal.

The fact that the rationale for a quality attribute requirement is always a business goal gives us a new lens through which to examine quality attribute requirements. By identifying business goals that are at work, we can ask which quality attributes would be important to help achieve the business goal and what aspect of the quality attribute contributes to that achievement. We can compare this list to the existing quality attribute requirements (if there are any) and see if there is a mismatch.

Not all business goals for an organization are achieved through the construction of a system. For example, "reduce cost" may come about by lowering the facility's thermostats in the winter or reducing employees' pensions.

Other business goals may directly affect the system without precipitating a standard quality attribute requirement per se. For example, we know of a case where a manager pressed for an architecture to include a database because the organization's database group was currently sitting idle. No requirements specification would capture such a "requirement." And yet that architecture, if delivered without a database, would be just as deficient from the point of view of the manager as if it had failed to deliver an important user function.

Figure 2 illustrates the salient points so far. In the figure, the arrows mean "lead to;" the solid arrows are the one highlighting the relationships of most interest to architects.

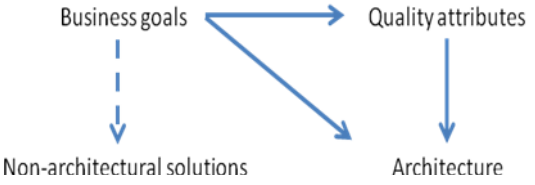


Fig. 2. Some business goals lead to quality attribute requirements (which in turn lead to architectures); others lead directly to architectural decisions without imposing quality attribute requirements; still others lead to non-architectural solutions.

4 A Canonical Set of Business Goals

The business literature provides a plethora of papers on business goals or business models for organization. We conducted a structured literature review using the Proquest ABI/Inform Global database, which covers over 3,000 business-oriented publications, as well as a simple Google search. In both cases, various combinations of "business goals," "business models," and "survey" or "studies" were used as search terms. Space limitations prevent us from reproducing the entire collection of business goals we harvested from the papers uncovered by our search; a full accounting is given in [5].

We conducted an affinity (clustering) exercise with the goals uncovered from our search of the business goal literature. The result is the set of ten business goal categories shown in Fig. 3.

One goal could easily fit in more than one category. In an elicitation method, the categories should prompt questions about the existence of organizational business goals that fall into that category. If the categories overlap, then this might cause us to ask redundant questions. This is not harmful and could well be helpful. SIMPLE is an example of how the second goal category ("Meeting financial objectives") can be analyzed.

1. Growth and continuity of the organization
2. Meeting financial objectives
3. Meeting personal objectives
4. Meeting responsibility to employees
5. Meeting responsibility to society
6. Meeting responsibility to country
7. Meeting responsibility to shareholders
8. Managing market position
9. Improving business processes
10. Managing quality and reputation of products

Fig. 3. A canonical set of business goals

Furthermore, Osterwalder and Pigneur [14] provide us with a perspective for discussing goal change over time. They identify five forces that will change over time and which need to be considered when establishing business goals. These forces are legal, social, technological, competitive, and customer. Any specific business goal has to be considered in light of how these five forces might cause it to change over time.

5 Expressing Business Goals

If business goals are important to inform a product line's business case as well as its architecture, it stands to reason that a way to articulate a business goal in a well-formed manner would be useful. Capturing business goals and then expressing them in a standard form will let them be discussed, analyzed, argued over, rejected, improved, reviewed – in short, all of the same activities that result from capturing any kind of requirement. To this end, we introduce business goal scenarios.

The purpose of a business goal scenario is to ensure that all business goals are expressed clearly, in a consistent fashion, and contain sufficient information to enable their processing through the further steps of our technique.

Similar to our own quality attribute scenarios [5], our business goal scenario has seven parts. They all relate to the system under development, the identity of which is implicit. Together, they provide a provenance for a business goal that will contribute to its understanding and its interplay with other goals. The seven parts are:

1. **Goal-source.** Who (or what artifact) provided the statement of the goal?
2. **Goal-subject.** This is the stakeholder who owns the goal and who wants it to be true. The stakeholder might be an individual, an individual in an identified organization if more than one organization is in play, or (in the case of a goal that has no one owner and has been assimilated into an organization) the organization itself. To express a business goal meaningfully, as well as capture information to resolve goal conflicts, we need to know the person who owns the goal. This we call the *goal-subject*. If the business goal is, for example, "maximize dividends for the stakeholders," who is it that cares about that? It is probably not the programmers, the system's end users, or (if an acquired system) anyone in the acquisition organization. Stakeholder theory [12] will help us identify the goal-subject(s) of a business goal, as well as help us identify people who might have business goals to contribute. We will seek stakeholders with high "salience" [12] from whom to elicit business goals, and record those stakeholders as the goal-subjects.
3. **Goal-object.** This is the entity to which the goal applies. All goals have goal-objects – we want something to be true about something (or someone) that (or whom) we care about. Seen in this light, the goals cataloged throughout this paper can all be re-elaborated by making their respective goal-objects explicit. By examining the collected goals and categories we observed in our literature survey, we were able to discern the following goal-objects: *Individual, System, Portfolio, Organization's employees, Organization's shareholders, Organization, Nation, and Society.* For example, for goals we would characterize as furthering one's self-interest, the goal-object is "Individual." For goals we would characterize as ethical, the goal-object can be "Nation" or "Society." For some goals the goal-object is

clearly the development organization, but for some goals the goal-object can be more refined, such as the organization's rank-and-file employees or shareholders. Goals with multiple goal-objects are common, but not harmful to the process of eliciting business goals.

4. **Environment.** This is the context for this goal. It acts as a rationale for the goal. A good source for this entry is the five different environmental factors of Osterwalder and Pigneur [14] (social, legal, competitive, customer, and technological). The goal's environment captures what is relevant to the goal in each of these five encompassing areas.
5. **Goal.** This is any business goal able to be articulated by the person being interviewed.
6. **Goal-measure.** This is a measurement to determine how one would know if the goal has been achieved.
7. **Value.** This is how much the goal is worth. It might be expressed in terms of what might be willingly paid to achieve it, or a ranking against other goals in a collection or as a range such as "high, medium, or low".

6 A Method for Eliciting Business Goals

To elicit business goals, we crafted a seven-step method reminiscent of some of our other workshop-oriented stakeholder-centric architecture-focused methods such as ATAM [4]. ATAM relies on stakeholder participation and scenario generation to gather data. The elicitation method recognizes that much of the most valuable information is in the heads of the stakeholders rather than in documents and models, but that the stakeholders need to be guided through the process. Stakeholders want to share their information but they need a structured way to provide it.

A stakeholder-centric, scenario-based method requires an explicit criterion that can be applied to the sources of scenarios to ensure a complete survey of possible goals. For example, the CONOPS for the product line organization lists all the roles. Each relevant role provides scenarios. The scenarios must be prioritized since there are usually many more scenarios than time to analyze them. Finally, the method needs an intuitive analysis method so that it can be applied quickly.

Because an outcome of the business goal elicitation method is a set of quality attribute requirements with a pedigree rooted in business goals, we call our method the Pedigree Attribute eLicitation Method (PALM). The steps of PALM are:

1. **PALM overview presentation:** Overview of PALM, the problem it solves, its steps, its expected outcomes.
2. **Business drivers presentation:** Briefing of business drivers by project management. What are the goals of the customer organization for this system? What are the goals of the development organization? This is a lengthy discussion that gives the opportunity of asking questions about the business goals as presented by project management.
3. **Architecture drivers presentation:** Briefing by the architect on the driving (shaping) business and quality attribute requirements.
4. **Business goals elicitation:** Using the standard business goal categories to guide discussion, we capture the set of important business goals for this system. Business

goals are elaborated, and expressed as business goal scenarios. We consolidate almost-alike business goals to eliminate duplication. Participants then prioritize the resulting set to identify the most important ones.

After step 4, one of the purposes of PALM – namely, the capture of business goals to inform the business case – will have been discharged. The next two steps relate the business goals to quality attribute requirements to inform the product line architecture.

5. **Identifying potential quality attributes from business goals.** For each important business goal scenario**,** participants describe a quality attribute that (if architected into the system) would help achieve it. If the QA is not already a requirement, this is recorded as a finding.
6. **Assignment of pedigree to existing quality attribute drivers.** For each architectural driver named in Step 3, we identify which business goal(s) it is there to support. If none, that's recorded as a finding. Otherwise, we establish its pedigree by asking for the source of the quantitative part: E.g.: Why is there a 40ms performance requirement? Why not 60ms? Or 80ms?
7. **Exercise conclusion.** This constitutes a review of results, preview of next steps, and gathering of participant feedback.

PALM can be used to sniff out missing quality attribute requirements early in the life cycle. For example, having stakeholders subscribe to the business goal of improving the quality and reputation of their products may very well lead to (for example) security, availability, and performance requirements that otherwise might not have been considered.

PALM can also be used to inform the architect of business goals that directly affect the architecture without precipitating new requirements. For example, if an organization has the ambition to use a product as the first offering in a new product line, this might not affect any of the requirements for that product and therefore not merit a mention in that product's requirements specification. But this is a crucial piece of information that the architect needs to know early so it can be accommodated in the design.

PALM can also be used to examine particularly difficult quality attribute requirements to see if they can be relaxed. We know of more than one system where a quality attribute requirement proved quite expensive to provide, and only after great effort, money, and time were expended trying to meet it was it revealed that the requirement had no analytic basis, but was merely someone's best guess or fond wish at the time.

Finally, different stakeholders have different business goals for any individual system being constructed. The acquirer may want to use the system to support their mission; the developer may want to use the system to launch a new product line. PALM provides a forum for these competing goals to be aired and resolved. This is especially useful in the early stages of a product line, when the business case is being crafted.

7 Experience with PALM

We applied PALM to a system being developed by Boeing's Air Traffic Management unit. To preserve confidentiality, we will call this system The System Under Consideration (TSUC) and summarize the exercise in brief.

TSUC will provide certain on-line services to the airline companies to help improve the efficiency of their fleet. Thus, there are two classes of stakeholders for TSUC – Boeing and the airline companies. The stakeholders present when we used PALM were the chief architect and the project manager for PALM.

Some of the main goals that were uncovered during this use of PALM were:

- Impact of the system both on the user community and the developer community
- TSUC was viewed as the first system in a future product line.
- The lifetime of TSUC in light of future directions of regulations affecting air traffic.
- The possibility of TSUC being sold to additional markets.
- Governance strategy for the TSUC product.

The exercise helped the chief architect and the project manager share the same vision for TSUC, such as its place as the first instance in a product line and the architectural and look-and-feel issues that flow from that decision.

The ten canonical business goals ended up bringing about discussions that were wide ranging and, we assert, raised important issues unlikely to have been thought of otherwise. Even though the goal categories are quite abstract and unfocussed, they were successful in triggering discussions that were relevant to TSUC. The result of each of these discussions was the capture of a specific business goal relevant to TSUC.

8 Related work

Two related communities already use scenarios to capture business goals: The Architecture Development Method of The Open Group Architecture Framework (TOGAF) [17] includes a treatment of business scenarios, which is their vehicle for capturing and expressing business goals..

Techniques such as goal-oriented requirements engineering [3][18] are well-positioned to gather and manage the relation between goals and requirements but goal-oriented requirements engineering methods tend to be heavyweight and do not provide a canonical set of business goals from which to begin.

Kazman and Bass [11] presented an early study in which a categorization of business goals was created. The categories were formed by looking at the business goals derived from twenty-five applications of the Architecture TradeOff Analysis Method (ATAM). These were the starting point for the categorization developed for PALM.

Nord et al. described how to integrate the ATAM and CBAM techniques. The CBAM and ATAM techniques both elicit scenarios form stakeholders. Many of these reflect the business goals of the organization for which the architecture is designed.

Evellin et al. [9] reported that using a catalog of non-functional requirements during business goals elicitation resulted in the identification of some goals that had not been discovered. They did not relate these to the software architecture. Rosen [16] explains the importance of business analysis, and hence business goals, to the success of a project. He lists questions that a software architect should have in mind when participating in the development of a business model.

PALM provides a more comprehensive approach than other techniques. It is not a complete business analysis method but it is complete with respect to the elicitation of business goals and is lighter weight than many of the goal-oriented requirements engineering methods.

9 Conclusions

Systematically eliciting and capturing business goals for a system is eminently useful in crafting a product line's business case as well as its architecture. PALM is a method for doing so. For the product line architect, PALM goes on to tie the business goals to quality attribute requirements that would lead to systems that satisfy those goals. PALM can help identify quality attributes that are missing, or are superfluous or too restrictive.

Business goals are not always written down in an accessible fashion; many times they're implicit and have never been identified. The business goal categories at the heart of PALM serve as elicitation aids or "conversation starters." You can use these categories to draw out an organization's specific goals (if any) in each category, using the business goal scenario format to capture them. Then you can ask how the strategy being explored in the business case (such as adopting the product line approach) could contribute to each goal using SIMPLE to help make quantitative judgments.

PALM was applied to an industrial system in the air traffic domain and helped elicit specific quality attribute requirements that might otherwise have been overlooked. The architecture community has identified architecturally significant requirements as those requirements that will most heavily influence the design of the architecture. Architecturally significant requirements must support high priority business goals or there is no reason to focus on them when designing the architecture. The output of PALM becomes immediately useful to the architect.

References

[1] How to Develop an IT Business Case. RMS Business Skills Training. Resource Management Systems, Inc., New York (2003), http://www.rms.net

[2] A Framework for Software Product Line Practice, Version 5.0, http://www.sei.cmu.edu/productlines/frame_report/index.html

[3] Anton, A., McCracken, W., Potts, C.: Goal Decomposition and Scenario Analysis in Business Process Reengineering. In: Wijers, G., Wasserman, T., Brinkkemper, S. (eds.) CAiSE 1994. LNCS, vol. 811. Springer, Heidelberg (1994)

[4] Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, Reading (2003)

[5] Clements, P., Bass, L.: Relating Business Goals to Architecturally Significant Requirements for Software Systems (CMU/SEI-2009-TN-026). Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2009) (forthcoming)

[6] Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading (2002)

[7] Clements, P.C., McGregor, J.D., Cohen, S.G.: The Structured Intuitive Model for Product Line Economics (SIMPLE) (CMU/SEI-2005-TR-003). Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2005)
[8] Cohen, S.: Case Study: Building and Communicating a Business Case for a DoD Product Line (CMU/SEI-2001-TN-020, ADA395155). Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2001)
[9] Evellin, C.S., Cardoso, J., Almeida, P., Guizzardi, G., Renata, S.: Eliciting Goals for Business Process Models with Non-Functional Requirements Catalogues. In: Halpin, T., Krogstie, J., Nurcan, S., Proper, E., Schmidt, R., Soffer, P., Ukor, R. (eds.) Enterprise, Business-Process and Information Systems Modeling. LNBIP, vol. 29 (2009)
[10] Jensen, P.: Experiences with Product Line Development of Multi-Discipline Analysis Software at Overwatch Textron Systems. In: Proceedings of the 11th International Software Product Line Conference, Kyoto, Japan, September 10-14, pp. 35–43. IEEE Computer Society, Washington (2007)
[11] Kazman, R., Bass, L.: Categorizing Business Goals for Software Architectures, CMU/SEI-2005-TR-021 (2005)
[12] Mitchell, R.K., Agle, B.R., Wood, D.J.: Toward a Theory of Stakeholder Identification and Salience: Defining the Principle of Who and What Really Counts. The Academy of Management Review 22(4), 853–886 (1997), http://www.jstor.org/stable/259247
[13] Nord, R., Barbacci, M., Clements, P., Kazman, R., Klein, M., O'Brien, L., Tomayko, J.: Integrating the Architecture Tradeoff Analysis Method (ATAM) with the Cost Benefit Analysis Method (CBAM), CMU/SEI-2003-TN-038, Software Engineering Institute (2003)
[14] Osterwalder, A., Pigneur, Y.: An ontology for e-business models. In: Currie, W. (ed.) Value Creation from e-Business Models, pp. 65–97. Butterworth-Heinemann, Oxford (2004)
[15] Porter, M.E.: Competitive Strategy. Free Press, New York (2004)
[16] Rosen, M.: The Role of Architecture in Business Analysis, http://msdn.microsoft.com/en-us/library/bb508953.aspx
[17] The Open Group, The Open Group Architecture Framework (TOFAF), Version 9, http://www.opengroup.org/togaf/
[18] van Lamsweerde, A.: Goal-Oriented Requirements Engineering: A Guided Tour. In: Proceedings, RE 2001, 5th Intl. IEEE Symposium on Requirements Engineering, Toronto, pp. 249–268 (August 2001)

Aligning Business and Technical Strategies for Software Product Lines

Mike Mannion[1] and Juha Savolainen[2]

[1] Glasgow Caledonian University, 70 Cowcaddens Road, Glasgow, G4 0BA, UK
m.a.g.mannion@gcu.ac.uk
[2] Nokia Research Center, Itämerenkatu 11-13, 00180 Helsinki, Finland
juha.e.savolainen@nokia.com

Abstract. A successful software product line strategy has business goals, a business strategy, a target market and a technical strategy that is aligned with the business goals and the target market. A common challenge in a number of organizations is for business and engineering units to understand what business and technical strategy alignment actually means in practice and to maintain that alignment as business goals and target markets evolve. If they are misaligned, then at best significant development inefficiencies occur, and at worst there is loss of market share. This paper explains different business and technical strategies, describes commonly used engineering techniques to manage commonality and variability and their deployment under different strategies.

Keywords: Product lines, business alignment, business strategy, feature modeling, software architecture.

1 Introduction

A successful software product line strategy has business goals, a business strategy, a target market and a technical strategy that is aligned with the business goals and the target market. A common challenge in a number of organizations is for business and engineering units is to understand what business and technical strategy alignment actually means in practice and to maintain that alignment as business goals and target markets evolve. If they are misaligned, then at best significant development inefficiencies occur, and at worst there is loss of market share. This paper is aimed at those who have responsibility for the delivering the business and technical strategies and to help them collaborate at the interface between the two.

In this paper we explain which technical strategies are broadly aligned with which business strategies. Our view is that there are both technical and political factors that can have an effect on this alignment. In the next section we outline business and technical strategies. Section 3 describes several commonly used engineering techniques to manage commonality and variability and discusses their deployment under different technical strategies. Section 4 discusses these ideas in the context of related work. This paper is partially based on our tutorial on aligning business and technical strategies [14].

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 406–419, 2010.

2 Software Product Line and Technical Strategies

A software product line business strategy focuses on commercial goals, products to be developed, markets for the products and financial, marketing and sales, engineering, and customer support strategies to underpin those choices. A corresponding technical strategy is focused on product software development and product derivation processes, software architecture choices, design, implementation and test methods to underpin those choices.

Good software product line engineering is managing commonality and variability of a product line efficiently and effectively as it evolves. Each product is formed by taking applicable assets from a base of common assets, tailoring them as necessary through preplanned variation mechanisms, adding any new components that may be necessary, and assembling the collection according to the rules of a common, product-line-wide architecture. In theory, building a new product is more assembly than creation using a predefined guide that specifies the exact product-building approach. In practice new products not only require new components but also new variation mechanisms to be created either for the new components or to access existing components in a different way. This requires a deep insight into the product line and a skillful blend of engineering and management.

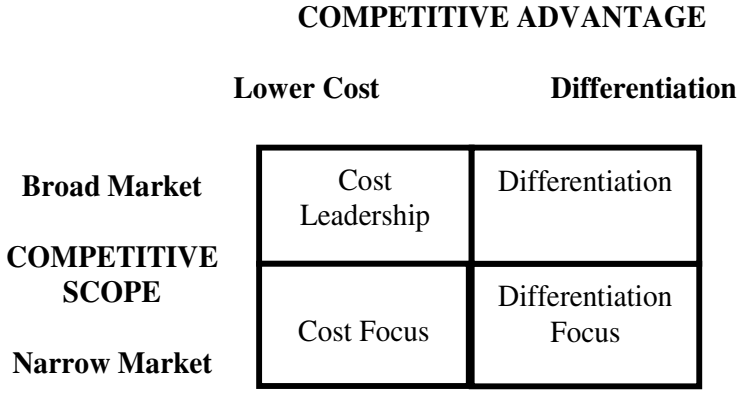


Fig. 1. Competitive strategies

Michael Porter [1] identified 4 primary generic competitive business strategies across two axes, market scope and competitive advantage (Figure 1). With a Cost Leadership strategy an organization targets a broad market and keeps costs lower than its rivals either selling below or at average industry prices. To implement a cost leadership strategy is to reduce the feature set of the product, improve process efficiencies, gain unique access to a large source of lower cost materials or make optimal outsourcing and vertical integration decisions.

With a Differentiation strategy an organization targets a broad market with a product or service that has a unique combination of attributes that is valued by customers more than competitors' products, allowing a premium price to be charged, which will

more than cover the extra costs incurred in offering the unique product. A differentiation strategy is implemented by having a well-founded corporate reputation for quality and innovation and a strong sales team with the ability to successfully communicate the perceived strengths of the product. Sometimes these strategies are stretched either because an organization floods the market with a large number of similar products relying on its brand name for sales rather than customer understanding of product differences, or there is a drive towards mass-customization where each customer receives a personalized copy of the product.

With a Cost Focus or Differentiation Focus strategy an organisation targets a narrow market segment and within that seeks cost advantage or differentiation. It implements this strategy by tailoring a broad range of product development strengths. The strategy can generate high customer loyalty and act as a barrier to entry for competitors. Organisations have lower volumes and therefore less bargaining power with their suppliers although organisations pursuing a differentiation-focused strategy may be able to pass higher costs on to customers since close substitute products do not exist.

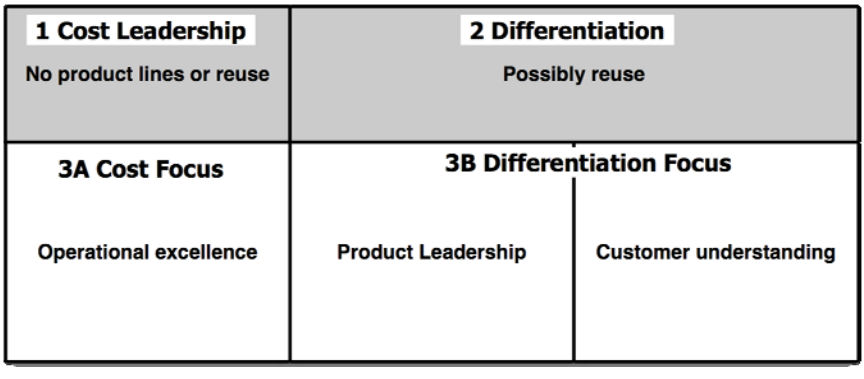


Fig. 2. Business Strategies and Technical Strategies

Figure 2 shows a set of technical strategies that are commonly aligned with product line strategies: no reuse, possibly reuse, operational excellence, product leadership and customer understanding (we have borrowed the last three terms from Tracy & Wisrsema's types of value discipline [2]).

When the primary generic competitive strategy is Cost Leadership the emphasis for the product development team is on maximizing the reuse of common assets more than permitting variation. New products are typically constructed by small changes to an existing product. Over a period of time a new product assumes its own maintenance trajectory separate from other products, there is little or no product line or reuse, and the technical strategy is effectively single-system development with some reuse.

Under Cost Focus, a small set of product features are tailored to the needs of the market segment. A common technical strategy is to concentrate on making the product **operationally excellent** through the development of a common reference architecture, a set of components that are used in each product and few variations. This confines the locus of changes, reducing costs and delivering value-for-money.

When the primary generic competitive strategy is Differentiation the emphasis is on adding product feature combinations (at competitive prices) that rivals do not have i.e. permitting variation rather than focusing on reuse. A common technical strategy is product-specific architecture-centric, high variation management, asset-based reuse. Under Differentiation Focus the shape of this technical strategy will be a function of the amount of variation to be managed. In **product leadership** the focus is on delivering feature variations that enable the product line to maintain its brand image as a market innovator and hence leader. In **customer understanding** the focus is on creating customized product variants. At the extreme of product leadership, an organization may simply generate a large number of product variants that are actually very similar but overwhelm the market, and rely more on the brand of the organization for sales and less on understanding the differences across the product set. At the extreme of customer responsiveness is mass-customization in which the goal is to create customized product variants and each customer receives a personalized copy of the product. At both extremes variability management becomes less centralized and each product has its own configuration of components.

3 Techniques for Managing Commonality and Variability

A number of techniques for managing commonality and variability exist. The choice of technical strategy within which these techniques are deployed has a clear effect on the characteristics of this deployment. This section describes 3 popular techniques for managing commonality and variability and discusses their characteristics under the 3 different technical strategies shown in Figure 2.

3.1 Management of Commonality

Commonality provides the basis for reuse. Similarities among a set of products permit economic benefits to be achieved by sharing common parts between products. Whilst the identification of commonality often appears to be straightforward, its engineering must be carefully managed. Over time what is common can become variable and what is variable can become common. In [3] we categorized features into three different categories based on how they are shared among products:

- **Common** features that are required by all product line variants
- **Partial** features that are required by some of the product line variants
- **Unique** features that are requirement by exactly one product line variant

The categorization of the wanted product selections provides the basis for *required variability*. However, depending on the business strategy the required variability is differently converted to *provided variability* that is represented by the feature types, which constrain how the variability can be realized. Operational excellence tries to reduce costs by increasing commonality, customer understanding intends to increase differentiation by increasing variability, and product leadership aims to balance the commonality and variability aspects.

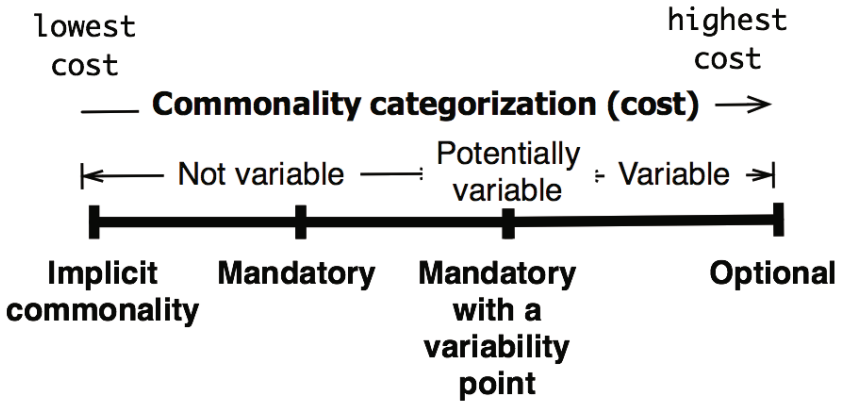


Fig. 3. Commonality categorization with increasing cost towards right [4]

Previously [4] we showed that typical product lines have many levels of commonality that differ from their evolution characteristics and the overall cost of commonality to the product line. Figure 3 shows a commonality categorization.

The least cost form of commonality is one that is not modeled or managed. We use the term *implicit commonality* for this purpose i.e. commonality not shown in feature trees or any other artifacts because it is not managed centrally and used as is. Commonality can be also managed as mandatory features. Mandatory features require effort during the analysis process and their management in a feature model is more costly then implicit commonality.

Considerable costs are incurred if mandatory features are attached to a variation point. A variation point is sometimes needed if a feature has been variable before or is potentially variable. Variability points are expensive, partly because the mechanisms to handle the potential variability must be analyzed, designed, and implemented. Optional features represent variability, but can also support commonality. An optional feature can be common to some products, thus expressing commonality among the set of products that share this feature.

When the technical strategy is **operational excellence** all common features are reused always together. If a unique feature is highly valuable for a product, it is developed independently for the product, and not considered for reuse. If a unique feature is not highly valuable then it is ignored and not implemented. If a partial feature is highly valuable for a product, it is developed and reused in all products. If a partial feature is not highly valuable then it is also ignored and not implemented. As a consequence, some products will have these features even if they did not originally require these features. Under operational excellence most features have implicit commonality to minimize cumulative costs from analysis, development and derivation. Mandatory features with variation points should be avoided because all features should be either strictly mandatory or optional and used only by one product.

When the technical strategy is **product leadership** all features are viewed as potentially reusable. Common features shared by all products become mandatory features. New features are typically introduced as unique, optional features in order to maximize the revenue from these new features, but shortly shared with a number of

products increase volumes by adding the new feature to new price points. Over time, partial features tend to dominate as most features are common to some products (*near commonality* [5]) and only a few features are common to all or unique to one. Under product leadership the focus is on the evolution of variability. Ideally product leadership tries to benefit from the differentiating value of new features by introducing them in limited products, but also to achieve low cost by sharing common features. In Figure 3 the selection property of most features drifts from right to left over time e.g. optional features tend to become mandatory.

When the technical strategy is **customer understanding** unique features are introduced to make products different from each other in ways that are valuable for users. This approach has an impact on the inclusion of partial features. Sometimes, a product may be prevented from including a feature that is already a differentiating feature for another product [6]. Common features are still allowed for cost reduction reasons. When using a customer understanding strategy, a large amount of optional features are needed to realize the needed variations in the customer requirements. If this strategy is used to full effect then optional features are only combined with implicit commonality.

3.2 Feature modeling

Feature modeling is a popular approach to manage and visualize commonality and variability in a product line. There are a number of ways to perform feature decomposition. Whilst functional decomposition is very popular other approaches can be used to determine or at least influence decomposition. These include the infrastructure (e.g. structure, interfaces, platform, licenses) of available software and hardware assets; the organizational structure and staff capability, and the business strategy adopted by the organization for the product line, the product line's relative importance to the organisation's business and the corresponding resource priorities that are allocated. Over the lifetime of a product line several factors interact with each other.

While a number of different feature modeling methods have been published, in this paper we use simple feature trees to visualize various effects that different business strategies may have on the structure and variability in the feature models. The notation is the same as used in our previous work [7]. Mandatory features are always selected if their parents are selected (in all examples we assume this). An optional feature may or may not be selected. From a set of alternative features, one must select one and only one. From a set of multiple features, at least one must be selected but more than one can be selected.

Given a product line model of features, it is possible to select the features for a new instance of the product line from this model. There are two primary methods of selection of features [8]. *Free selection* allows an engineer to browse a product line model and simply copy and paste a single feature from anywhere in the model to the new product features. It does not use the constraints built into the model structure to guide model traversal and can lead to selection combinations being made that do not satisfy the constraints of the product line model, e.g. two mutually exclusive features can be selected or features that must be included can be omitted. In addition there can be an untenable number of choices making single product specifications time-consuming to select and check that they are free of errors.

An alternative method to free selection is *constraint-based selection*, which is grounded in using the constraints built into the product line model to drive selection and permits choices to be made only at variation points. This ensures that the choices produce a set of single product features that satisfy the constraints built into the product line model and reduces the time spent on specification. Before the feature selection process begins, the values of global parameters are defined and all unavailable features and their descendents are made not selectable. Then starting from the root of the feature model, each tree is traversed depth-first and selected features are added to the new product. During traversal not every feature will be visited. Visitation will depend on prior selection.

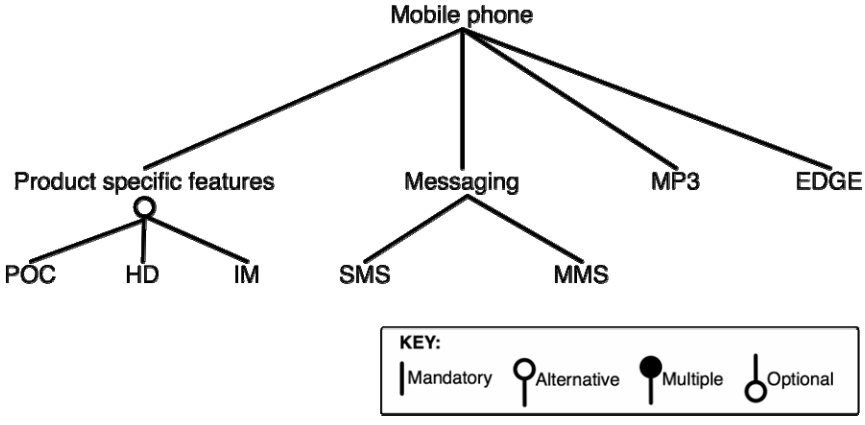


Fig. 4. A feature model for a product line using operational excellence business strategy

Since **operational excellence** targets implicit commonality, the modeling of mandatory features in a feature tree is not typically used. Rather a simple list of features is sufficient. However, to demonstrate the differences in business strategies the example in Figure 4 is shown. In Figure 4, all other features except the product specific features are mandatory. At the extreme, this strategy supports only a single set of alternative features from which each product can choose one and only one. In most product lines this extreme is rare and most products can have more than one unique feature. Typically, no selection of features is required when utilizing operational excellence. All mandatory features are reused by default and product specific features are developed by the product programs. However, if more then one unique or even some partial features are allowed then constraint-based selection may become useful.

Under **Product leadership** all possible variability is in the feature model and there is an assumption that this variability will change during the product line evolution. The intention is to realize all reuse opportunities, also in partial features.

To simplify the constraint-based selection process a default value is a property that can be attached to an optional feature to indicate whether the feature is by default selected or not. A feature that is selected by default is called excludable (i.e. it does not have to be explicitly selected to be included in the new product) and one that is by

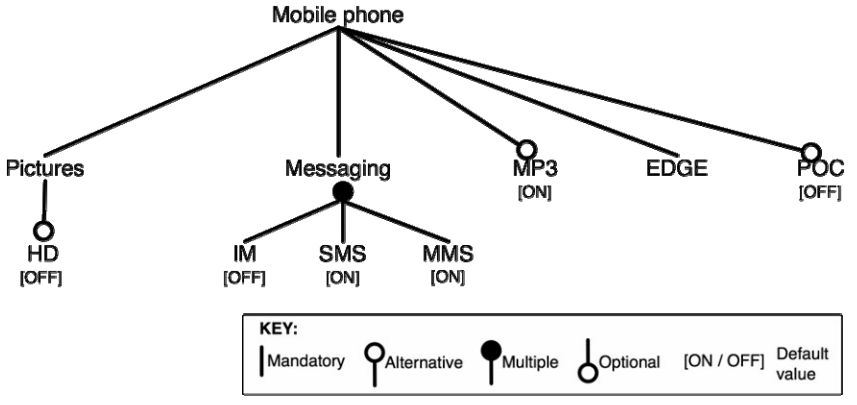


Fig. 5. A feature model for a product line using product leadership business strategy

default not selected includable (i.e. it has to be explicitly selected to be included in the new product). In Figure 5, excludable features are tagged with [ON] and includable features with [OFF].

By a careful use of default values, one can decrease the effort needed in selecting features during derivation. Each time one needs to override a default value, additional effort is naturally needed. Default values can be attached to more complex variability types. One feature from a set of mutually exclusive features can be excludable whereas others can be includable. In addition, a set of multiple features (must select one, but can select many) may contain many excludable features. Products will select features using constraint-based selection and default values will make that selection efficient.

A company that uses a **customer understanding** strategy may not use feature trees at all if the number of constraints on the variability is low. If a feature tree is used, then it is typically dominated by optional and multiple features. Default values are normally not very useful since products do not share similar selections, because they can be very different and try to prevent unnecessary sharing of features. Free selection is the most appropriate method for products to select features.

3.3 Feature Dependency Management and the Software Architecture

Feature dependency management deals with the functional and implementation dependencies among features. These dependencies have a major impact on how features can be realized through software architecture. When deploying features to the architecture, the dependencies between features become the dependencies between architectural elements.

A typical approach to achieve **operational excellence** is through architecture using a product platform approach for product lines. All common features are placed in the platform and reused together in all the products. Products may independently create differentiating functionality on top of the platform. Figure 5 shows a mobile phone product platform in which the common features are SMS, an MP3 players, Multimedia

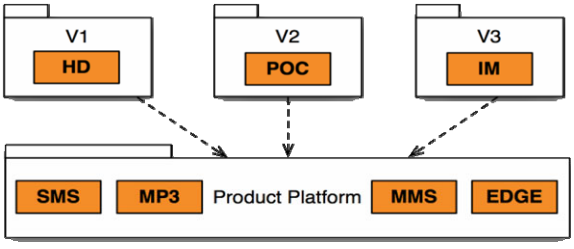


Fig. 6. Platform architecture

Services and EDGE (fast data transfer on 2G networks). However each of products V1, V2 and V3 require distinctive features. The reader should note that a typical product platform would have many more features as a part of the platform compared to number of variable features than what is described in this example.

Product platforms are used to maximize the coverage of common features, which are always selected and reused by all the products. Only those features that provide a great incentive for individual products will be independently developed beyond the platform.

One consequence of a platform-based approach is that reuse opportunities from partial features are typically ignored or undervalued. One way of reusing partial features is to introduce optional features to the platform. The benefits of the platform approach start to reduce rapidly after introducing optional features to the platform. Eventually this may lead to abandoning the platform approach for its alternatives. A technical decision to abandon the platform approach must be aligned with the business strategy. Otherwise the only result will be increased cost of developing products without any benefits.

Since the platform is always used as is the product derivation costs are minimal. Another reduction in cost happens because dependency management in the platform approach is easy. In theory, there are no restrictions on dependencies within the platform. The only restriction is that the product can depend on the platform, but the platform cannot depend on the products.

Having no restrictions on the dependencies within the platform allows very efficient implementation of new features, potentially reducing the overheads that various abstraction layers, hierarchical component invocations could be. However, if the functionality of the platform needs to be evolved often then costs may rapidly increase. Therefore, typically e.g. circular dependencies should be avoided also within the platform and stable interfaces should be aimed. This would also make easier to add new functionality to the platform itself.

Product leadership requires an architecture that can handle the variability while still getting the benefits from reuse. The chosen architecture depends often on the background of the product line. If the company has the background of platform based development then it is possible to try to extend the approach by introducing either more layers or components on top of the basic product platform.

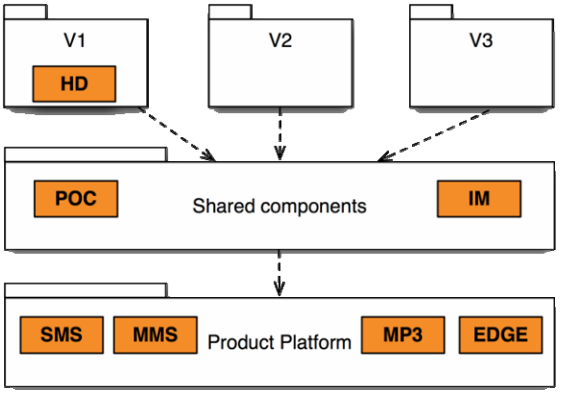


Fig. 7. Platform architecture evolving towards shared asset based reuse

Figure 7 shows a layer for shared but not common-to-all assets introduced on top of the product platform. Components in this new layer are selected by the products based on the features that these components support and most products (V2 and V3 in Figure 7) are built only by making these selections. Some products (V1 in Figure 7) may still create product specific functionality on top of the shared components and the platform. When the number of products increases and the scope of the product line is enlarged the common product platform may disappear and all components become centrally shared assets from which products may make selections.

Feature dependency modeling is driven by ability to make selections i.e. each product gets all the features that it has explicitly selected and nothing else. Not having excess features is especially important in embedded products that are mass-produced, where the bill of materials can be the deciding factor in the success of the product line. The limitations on how shared components can be connected together both restricts the selections as well as reduces the overall cost since the architecture can be based on some stable ground.

When the number of desired combinations of features increases and the number of truly common features becomes very small, a change to compositional development may be a choice. A shared asset-based approach with centralized variability management seems to work best when combined with a good reference architecture. If the product selections do not cluster, then creating a common reference architecture may be close to impossible. This can happen because the company uses a **customer understanding** business strategy. That is, the whole intent is to exactly match the customer needs even if it limits the ability to reuse or hinders the ability to create efficient a reuse infrastructure. Ultimately this may lead to stopping reuse altogether if no such low value elements exist that do not negatively affect the differentiation. In practice, we have not seen examples that have completely ignored reuse after starting using the product lines.

Table 1. Summary

Engineering technique	Operational Excellence	Product Leadership	Customer Understanding
Management of commonality	Mostly common features	Mostly partial features	Mostly unique features
Feature Modelling	Graphical model of small number of variable features. Product derivation by constraint-based selection.	Graphical model of large number of variable features (with default values). Product derivation by constraint-based selection.	No graphical model - too much variability. Product derivation by free selection.
Feature Dependency Management & Software Architecture	Common reference architecture needs manage little variability.	Common reference architecture needs manage a lot of variability.	No common reference architecture

Table 1 summarises the combination of engineering techniques and their properties under different technical strategies. The list of engineering techniques is illustrative based on our experience but is not exhaustive. Other techniques which may be used by an organization can be added as appropriate. The choice of technical strategy within which these techniques are deployed has a clear effect on the characteristics of this deployment. In turn, the actual deployment of a given engineering technique can be compared to what one might expect for a given technical strategy. Such a table can be valuable when reviewing overall project progress, reflecting on the alignment of the business and technical strategies and understanding whether the choice of engineering technique and its deployment are broadly in alignment with the choice of technical strategy. If there is significant deviation then the table can also be used as a tool to support the evaluation of whether the deviation is significant and what the impact is in terms of the risk of misalignment, whether it needs to be managed and how. Misalignment can and does occur over time for several reasons including: the development team not always being clear about the business strategy through poor or infrequent communications, new emergent engineering tools and technologies being deployed within a technical strategy that have a detrimental impact on existing product line solutions, and loss of product line expertise as senior staff changes occur as people change jobs. In addition, based on our experiences, one of the most frequent sources of misalignment is the previous success in software reuse in a different part of the organization. If one particular technical strategy is successfully used once, there is a organizational tendency to deploy that strategy again --- the assumption being that since it worked well last time, there appears to be no reason why it will not work again. This is fine if the business strategy used for all these different product lines is identical. However, if the business strategy is different, then misalignment is likely to occur.

In addition, apart from the deep technical issues that can cause business and technical strategies to be misaligned there are also some broader political factors that have

to be regularly monitored and managed. In previous work [13] we identified two examples in which the manipulative and often subtle manipulation of the technical strategy can cause a significant drift from the intended business strategy. First, tyranny of reuse occurs when maximizing reuse is the dominant technical priority regardless of what might be the best solution for a particular product or the product line. Emphasis is placed on ensuring all products use every reusable component, common user interface guidelines, standard interfaces leading to a common structure and behaviour to all products, and differentiation between the products is reduced. Second, local product optimization occurs when a product manager or product team priorities its own product or set of products over the corporate business needs. Within a very broad product line, e.g. a mobile phone, there are often sub-product lines of particular products, each with its own product manager and each manager wants to have a successful product. If performance success is measured by the number of products sold and this measure singularly dominates managerial performance evaluations, then this can lead to a poorly optimized product portfolio.

In presenting these ideas we recognize that there are a number of outstanding challenges including:

- the choice of business strategy types may need to be refined to suit different product line organizations
- the list of factors we have identified is not exhaustive and there will be other technical, political, financial and social ones
- what types of misalignment are there, what are the risks of misalignment and how can they be managed ?
- what combinations of factors appear to have a greater impact than others on ensuring alignment ?
- when, how, with what frequency should strategy alignment be evaluated ?
- what practical and realistic steps can be taken to manage misalignment?

4 Related Work

In [9] the influence of a number of key business factors in managing a successful software product line was demonstrated empirically through a quantitative survey of software organizations involved in the business of developing software product lines over a wide range of operations, including consumer electronics, telecommunications, avionics, and information technology.

In [10] a software product line engineering evaluation framework was presented that consisted of four dimensions: business, architecture, organisation and process. The purpose of the framework is to generate an evaluation profile that can serve as benchmark for effective software product line engineering, support capability evaluations of software production teams and aid the development of software product line engineering improvements plans. Zubrow and Chastek [11] listed a number of evaluation categories that are of interest to a product line manager, a core asset development manager, and a product development manager. Under each category, a broad set of measures is defined that returns information about performance (measuring cost, schedule, and quality of product efforts), compliance (measuring the adherence

of the product line effort to established procedures and processes), and effectiveness (characterizing how the overall product line effort is meeting its goals).

In [12] a production strategy describes how product line practices should be employed so that a product line organization will achieve its production goals. A production strategy is derived from an organization's business strategy. It is generated by eliciting and documenting an organization's requirements for the production system as a series of scenarios, identifying the production factors critical to the success of the organization's product line, identifying strategic actions that will address those critical factors, refining the strategic actions into a coherent strategy based on established business strategy development techniques, and strategy evaluation.

Whilst these broad frameworks are very useful contributions to they do not drive down into the detail of how the choice of a particular business strategy can affect or be affected by different engineering techniques that are used for managing commonality and variability. Our experience is that as a product line evolves this level of practical detail becomes important as tangible evidence for understanding, evaluating and managing the extent to which business and technical strategy alignment are in place.

5 Conclusion

A successful software product line is usually one in which there are clear business goals, a business strategy, a target market and a technical strategy that is aligned with all of these. This paper explains different business and technical strategies and shows how they should be broadly aligned. The paper then describes some commonly used engineering techniques to manage commonality and variability, and then explains how these techniques should broadly be deployed under the different technical strategies. The subsequent mapping between technical strategy and engineering technique can be a useful framework for evaluating whether the choice of business strategy and the implementation of a selected technical strategy remain aligned. If they are misaligned, then at best significant development inefficiencies occur, and at worst there is loss of market share.

References

1. Porter, M.: Competitive Strategy: Techniques for Analysing Industries and Competitors. The Free Press, New York (1980)
2. Treacy, M., Wisrsema, F.: The Discipline of Market Leaders: Choose Your Customers, Narrow Your Focus, Dominate Your Market. Perseus Books, Cambridge (1997)
3. Savolainen, J., Kuusala, J.: Consistency Management of Product Line Requirements. In: Proceedings of 5th IEEE International Symposium on Requirements Engineering (RE 2001), Toronto, Canada, August 27-31, pp. 40–47. IEEE Computer Society, Los Alamitos (2001) ISBN 0-7695-1125-2
4. Savolainen, J., Bosch, J., Kuusela, J., Männistö, T.: Default Values for Improved Product Line Management. In: Proceedings of the Software Product Line Conference (SPLC), pp. 51–60 (2009)
5. Lutz, R.R.: Toward Safe Reuse of Product Family Specifications. In: Symposium on Software Reusability (SSR 1999), pp. 17–26. ACM Press, New York (1999)

6. Savolainen, J., Kauppinen, M., Mannist, T.: Identifying Key Requirements for a New Product Line. In: Proceedings of 14th Asia-Pacific Software Engineering Conference (APSEC 2007), Nagoya, Aichi, Japan, December 04 (2007) ISBN: 0-7695-3057-5
7. Ferber, S., Haag, Savolainen, J.: Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 235–256. Springer, Heidelberg (2002)
8. Mannion, M., Kaindl, H.: Using Parameters and Discriminants for Product Line Requirements. Systems Engineering 11(1), 61–80 (2008)
9. Ahmed, F., Fernando Capretz, L.: Managing the business of software product line: An empirical investigation of key business factors. Information and Software Technology 49, 194–208 (2007)
10. van der Linden, F., Bosch, J., Kamsties, E., Känsälä, K., Obbink, H.: Software Product Family Evaluation. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 110–129. Springer, Heidelberg (2004)
11. Zubrow, D., Chastek, G.: Measures for Software Product Lines (CMU/SEI-2003-TN-031). Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2003)
12. Chastek, G., Donohoe, P., McGregor, J.: Formulation of a Production Strategy for a Software Product Line, Technical Note (August 2009), CMU/SEI-2009-TN-025
13. Savolainen, J., Kuusela, J., Mannion, M., Vehkomäki, T.: Combining Different Product Line Models to Balance Needs of Product Differentiation and Reuse. In: Mei, H. (ed.) ICSR 2008. LNCS, vol. 5030, pp. 116–129. Springer, Heidelberg (2008) ISBN 978-3-540-68062-8
14. Savolainen, J., Mannion, M.: From product line requirements to product line architecture: optimizing industrial product lines for new competitive advantage. In: Proceedings of the 13th International Software Product Line Conference, p. 315 (2009)

Non-clausal Encoding of Feature Diagram for Automated Diagnosis

Shin Nakajima

National Institute of Informatics
Tokyo, Japan
nkjm@nii.ac.jp

Abstract. Automated support for finding unsatisfiable fragments is desirable to help removing deficiency in inconsistent feature diagrams. In encoding feature diagrams into propositional logic formulas, such a problem reduces to finding unsatisfiable cores. Standard algorithms work on clausal formulas, which looses the structural aspect of feature diagram. In this paper, we propose a new automated method, which employs a boolean constraint propagation algorithm for non-clausal formulas. The method can eliminate the problems in the previous approaches, where translation back and forth is required between feature diagram and clausal formulas.

Keywords: FODA Feature Diagrams, Unsatisfiability Checking, Non-clausal Formulas.

1 Introduction

Feature diagram, proposed in Feature Oriented Domain Analysis (FODA) [4], is a primary modeling notation in feature analysis. Various approaches to automated analysis of feature diagram have been studied [2]. M. Mannion [6] proposed the idea of connecting propositional logic formulas to feature diagrams. D. Benavides et al [2] transformed an invalid feature diagram into a Constraint Satisfaction Problem (CSP) in linear integer arithmetic domain, and solved it with Constraint Logic Programming (CLP). Such methods are now matured so that tool-supported automated analysis is possible, and one such tool is reported in [7]. Automated support for locating bugs in feature diagrams becomes one of the important issues [1, 8, 9].

In encoding feature diagrams into propositional logic, the consistency analysis problem turns to be satisfiability checking and the problem of locating bugs reduces to finding unsatisfiable cores in the formula. Standard algorithms, however, work on clausal formulas [5], which looses the structural aspect of feature diagram. Translation back and forth is required between feature diagram and clausal formulas. A lot of book-keeping is needed to extract unsatisfiable fragments from the conflict clauses.

In this paper, we propose a new method of automated bug location, which employs a boolean constraint propagation (BCP) algorithm for non-clausal formulas. We concentrate ourselves on the automated bug location problem. In

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 420–424, 2010.

particular, we discuss how the structural aspect and a piece of information obtained in the editing are used in designing the algorithm.

2 Propositional Logic-Based Analysis

In propositional logic-based methods, each primitive relationship is translated into a formula (γ^n) [2]. A whole feature diagram is a conjunction of such formulas; $\Gamma = \bigwedge_n \gamma_n$. The problem of consistency or validation checking turns to be that of satisfiability, $\models \Gamma$. It results in being unsatisfiable when the given feature diagram is invalid.

Generally, an unsatisfiable formula has a set of mutually inconsistent subformulas, namely unsatisfiable cores. A candidate to retract can be found from such cores. If the cores are minimal, bugs can be located in a pin-point manner. However, calculating the *minimal* unsatisfiable core for propositional logic formula is hard; it is a *coNP* problem, and hence, only heuristics-based approaches have been used to deal with it.

Standard algorithms for the check work on clausal formulas or Conjunctive Normal Form (CNF), and require a translation of Γ into CNF with a standard Tseitin's encoding method [5]. Although the translation is possible in principle,some drawbacks are known due to the semantic gap between the problem and CNF. The conversion entails a considerable loss of information about the problem's structure such as the graph view of feature diagram. Furthermore, extracting a piece of information needed for locating bugs becomes hard since lots of new propositional variables are introduced in the Tseitin's encoding.

3 Proposed Approach

3.1 Analysis Problem

In order to eliminate the problems in the previous approaches, we propose to use Boolean Constraint Propagation (BCP) algorithm for non-clausal formulas. The algorithm works directly on Γ, a conjunction of γ_n.

Each γ_n represents a fragment in the given feature diagram and thus Γ, not in CNF, faithfully reflects the problem. When some of γ_k's are identified in unsatisfiable cores, those corresponding features in the original diagram can be concluded readily in unsatisfiable fragments. Table 1 shows how each primitive relationship of feature diagram in [3] is represented in Disjunctive Normal Form (DNF). These can easily be extended to a super-feature that has more than two sub-features.

The translation from the standard form [2] to DNF is easily done by using equivalence relationships such as de Morgan's rules. Note that the translation is local in that each γ_n in the original formula is converted to γ'_n in DNF without affecting any other fragments; namely the structure is preserved.

$$\bigwedge_n \gamma_n = \bigwedge_n \gamma'_n, \qquad \gamma'_n = \bigvee_i (\bigwedge_j l^n_{i,j}),$$

The suffix n of $l^n_{i,j}$ denotes that such a DNF formula is obtained from a particular primitive n.

3.2 Constraint Propagation Algorithm

We present the algorithm for finding unsatisfiable core. Furthermore, we assume that Γ $(= \bigwedge_n \gamma'_n)$ is unsatisfiable. Such satisfiability or unsatisfiability can be ensured with the technique such as the one presented in [7].

Modern SAT solvers to implement extensions of DPLL algorithm introduce certain heuristics used in unit propagation [5]. It basically selects a candidate clause to which the current propositional values are tried to see if it is satisfied. Such heuristics are important because each propositional variable or an atom appears in many sub-formulas in the given large CNF. Contrary to such general cases, our problem can be simplified by making use of the graph structure of feature diagram.

In our encoding, a propositional variable appears only in a small number of γ'_ns, and those formulas are readily identified by considering the graph structural aspect. The propagation can be easy just to look at the features connecting with the edges in the graph.

The formulas are attached to each feature node in the diagram. Each node n has a formula α^n_j, β^n_i, and δ^n; α^n_j is a constraint condition corresponding to the attributes of n when it is a super feature of more than one sub-features, and β^n_i represents a composition rule. δ^n is true for a selected feature including the root and false for a de-selected one.

Figure 1 shows the propagation algorithm. It starts at a given start feature to traverse the graph in a breadth-first manner to find unsatisfiable formula γ'_n. Since the whole diagram Γ is a conjunction of γ'_n, the algorithm succeeds when it finds at least one such unsatisfiable formula.

The algorithm uses four global variables. Q is a FIFO-queue to store feature nodes to be explored in a breadth-first manner. V is a Set to contain features whose associated formulas have been already explored, and W is a FIFO-queue to maintain features whose formulas have been under-constraint. S is a Set to store feature nodes in unsatisfiable core.

In Figure 1, $formulas(P)$ returns a set of formulas to contain P. It first constructs a conjunct consisting of all such formulas, $\delta^P \wedge \beta^P \wedge \alpha^{super(P)} \wedge \alpha^P$, and translates it to a set of DNF formula (see Table 1).

Table 1. Disjunctive Normal Forms

(a) Mandatory	A0∧B0 ∨ ¬A0∧¬B0
(b) Alternative	A0∧B1∧¬B2 ∨ A0∧¬B1∧B2 ∨ ¬A0∧¬B1∧¬B2
(c) Or	A0∧B1 ∨ A0∧B2 ∨ ¬A0∧¬B1∧¬B2
(d) Optional	A0 ∨ ¬B1
(e) Opt. Alt.	A0∧¬B1 ∨ A0∧¬B2 ∨ ¬B1∧¬B2
(f) Opt. Or	A0 ∨ ¬B1∧¬B2
(g) Dependency	B2∧D1 ∨ ¬B2∧¬D1
(h) Exclusion	¬B2 ∨ ¬D1
(i) Implies	B2 ∨ ¬D1

```
Queue Q, W; Set S, V;
void unsatisfiable_core () {
  for (⟨p, v⟩ ∈ ν) {
    initialize(); f = propagate(p, v);
    if(f == UNSAT) break;
  }
  display_core ();
}

Status propagate(Prop p, Bool v) {
  enq(Q, ⟨p, v⟩);
  while (¬empty(Q)) { ⟨p, v⟩ = deq(Q); if (p ∈ V) then skip;
    put(S, p); add(ρ, p↦v); F = formulas(p); pending = false;
    for (γ ∈ F) {
      (ρ ⊨ γ)            → skip;
      (ρ ⊭ γ)            → return UNSAT;
      (ρ{q↦w} ⊨ γ)       → enq(Q,{⟨q, w⟩});        /* Unit Propagation */
      otherwise          → pending = true; enq(W,extract(ρ,γ));
    }
    if(¬pending) then put(V,p);
  }
  if(¬empty(W)) then return propagate(deq(W),true) else return NEXT;
}
```

Fig. 1. Constraint Propagation Algorithm

The main *for* loop checks satisfiability of each γ ($\in F$) under the bindings of ρ. It takes the *otherwise* branch when satisfiability of γ is not determined. Such γ will be queued to W to explore later. They are resumed when the algorithm reach a point where further unit propagation is not possible. Alternatively, when it detects unsatisfiability, propagation immediately returns, and *display_core*() displays all the feature nodes in S which belong to unsatisfiable fragments of the given inconsistent feature diagram.

3.3 Heuristics for Start Variable Selection

As in the case of DPLL-based SAT solver, the algorithm in Figure 1 is sensitive to the start variable in ν. The choice affects not only the execution time to detect unsatisfiability, but also results in different shapes of unsatisfiable cores. A certain heuristic method should be investigated.

In this paper, we assume an iterative process of constructing and debugging feature diagrams. We start with an original feature diagram Γ_{old} which has at least one valid configuration. Then, we edit the feature diagram to have a modified version, whose formula is denoted as Γ_{new}. In the editing process, some features together with their attributes are newly introduced while others may be modified. These changes can be collected on-line by GUI-based tool such as FD-Checker [7]. The set contains feature nodes only since a piece of information on their attributes is encoded as formulas and recovered from them.

Our problem is to find unsatisfiable fragments of the unsatisfiable feature diagram, sub-formulas γ'_ks, when we know that Γ_{new} is unsatisfiable. Since the editing process leads to unsatisfiability, we can safely guess that such fragments can be traced from the changes, from which we can obtain the appropriate start variables for ν in Figure 1.

4 Conclusions

We described a new algorithm to find unsatisfiable fragments in inconsistent feature diagrams, which is implemented in FD-Checker [7] so that we can compare it with the approach in [8]. The algorithm is designed to allow further heuristics. Selecting start variables is important from a viewpoint of obtaining fragment not too large nor too small for a good help in debugging. Finding appropriate heuristics for the selection is one of the future work.

References

1. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
2. Benavides, D., Segura, S., Ruiz-Cortes, A.: Automated Analysis of Feature Models 20 Years Later: A Literature Review. J. Information Systems (2010)
3. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison Wesley, Reading (2000)
4. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis Feasibility Study. CMU/SEI-90-TR-21 (1990)
5. Kroening, D., Strichman, O.: Decision Procedures. Springer, Heidelberg (2008)
6. Mannion, M.: Using First-Order Logic for Product Line Model Validation. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 176–187. Springer, Heidelberg (2002)
7. Nakajima, S.: Constructing FODA Feature Diagrams with a GUI-based Tool. In: Proc. SEKE 2009, pp. 20–25 (2009)
8. Nakajima, S.: Semi-Automated Diagnosis of FODA Feature Diagram. In: Proc. SAC 2010, pp. 2191–2197 (2010)
9. White, J., Benavides, D., Schmidt, D.C., Trinidad, P., Dougherty, B., Ruiz-Cortes, A.: Automated Diagnosis of Feature Model Configurations. J. Software and Systems (2010)

A Method to Identify Feature Constraints Based on Feature Selections Mining

Kentaro Yoshimura, Yoshitaka Atarashi, and Takeshi Fukuda

Hitachi Research Laboratory, Hitachi, Ltd.
7-1-1 Omika, Hitachi, Ibaraki 319-1292, Japan
{kentaro.yoshimura.jr,yoshitaka.atarashi.uw,
takeshi.fukuda.ge}@hitachi.com

Abstract. In this paper, we describe a novel method to identify constraints among features in a software product line, based on feature selections made in the past. Our approach takes feature selections of derived products as the input and extracts association rules between features such as "Products that selected feature i also selected feature j." We evaluated our method by applying it to a product line at Hitachi and identified 21 new constraints among 123 optional features.

1 Introduction

Once a software product line is established, it requires continuous evolution to react to market opportunities, changing technology trends, and competitors' new products [1]. Many successful practices, such as those for mobile phones [2] and automobiles [3], have increased the number of features to hundreds or even thousands due to continuous evolution.

Modeling feature constraints is a mechanism to reduce the number of theoretical feature combinations, and to guide engineers selecting features to specify a complete product [4] [5]. New features are introduced, and their constraints change as the product line evolves [6].

This paper addresses the problem of identifying new feature constraints in an evolving product line. We describe a novel method to recommend new feature constraints based on feature selections mining. Our approach takes feature selections made in the past as the input and computes association rules between features as recommendations for new feature constraints.

2 Our Method

2.1 Overview

Figure 1 shows an overview of our method. Our approach takes feature selections of a product line as the input and extracts association rules between features as recommendations of new feature constraints such as "Products that selected feature i also selected feature j." A user reviews each recommendation and accepts or rejects it. If the user accepts the recommendation, it is introduced into the core asset as a new feature constraint.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 425–429, 2010.

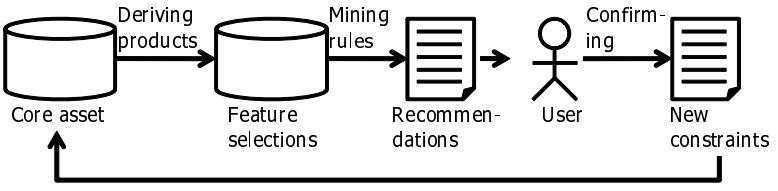


Fig. 1. Overview of our method

Table 1. Example of feature selections

Product ID	feature 1	feature 2	feature 3	feature 4	feature 5
1	✓			✓	
2		✓	✓		
3				✓	✓
4		✓	✓		
5		✓	✓	✓	
6		✓		✓	✓
7				✓	✓
8	✓			✓	

2.2 Association Rule Learning

This subsection describes the basic concept of our approach. Our approach applies association rule learning [7] to recommend new feature constraints in a large database of feature selections. Association rule learning is a well-known algorithm for mining relationships between sets of items in a large database of customer transactions, e.g., supermarkets transactions.

The problem of association rule learning is defined as: Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of n binary attributes called items. Let $D = \{t_1, t_2, \ldots, t_m\}$ be a set of transactions called the database. Each transaction in D has a unique transaction ID and contains a subset of the items in I. A rule is defined as an implication of the form $X \Rightarrow Y$ where $X, Y \subseteq I$, and $X \cap Y = \emptyset$. The sets of items X and Y are called the antecedent and consequent of the rule, respectively.

To illustrate the concepts, we use a small example from the feature selections shown in Table 1. The set of features is {feature 1, feature 2, feature 3, feature 4, feature 5} and the products containing the features ("✓" codes selection of a feature in a product) are shown in the matrix to the right. An example association rule for the product line could be {feature 2 ⇒ feature 3}, meaning that if feature 2 is selected, a product also requires feature 3.

2.3 Mining Feature Selections

Association rules can be supposed for all combinations between two features. To extract interesting rules from hypothetical ones, we calculate the threshold

measures such as support, confidence, and lift for all combinations of features, and extract rules with the minimum threshold measures.

The support $supp(X)$ of an itemset X is defined as the proportion of selections in the data set that contain the itemset. In the example product line, the support of a rule $supp$(feature 2 $\Rightarrow$ feature 3) is calculated as $3/8 = 37.5\%$.

The confidence of a rule is defined as the following equation.

$$conf(X \Rightarrow Y) = \frac{supp(X \Rightarrow Y)}{supp(X)} \tag{1}$$

For example, the confidence of the rule $conf$(feature 2 $\Rightarrow$ feature 3) is calculated as $supp$(feature 2 $\Rightarrow$ feature 3)/$supp$(feature 2) $= 0.375/0.5 = 75\%$.

The lift of a rule is defined as the following equation.

$$lift(X \Rightarrow Y) = \frac{conf(X \Rightarrow Y)}{supp(Y)} \tag{2}$$

For example, the lift of the rule $lift$(feature 2 $\Rightarrow$ feature 3) is calculated as $conf$(feature 2 $\Rightarrow$ feature 3)/$supp$(feature 3) $= 0.75/0.375 = 2.0$.

In this example, let us define the threshold measures supp, conf, and lift as 20%, 80%, and 2.0. Table 2 shows the extracted rules from all hypothetical rules in the example feature selections.

Table 2. Extracted rules with minimum thresholds: supp≥20%, conf≥80%, lift≥2.0

Rule ID	Antecedent	Consequent	Support	Confidence	Lift
1	¬feature 4	feature 3	25.0%	100.0%	2.7
2	feature 3	feature 2	37.5%	100.0%	2.0
3	feature 1	¬feature 2	25.0%	100.0%	2.0
4	¬feature 4	feature 2	25.0%	100.0%	2.0

3 Case Study

3.1 Case Study Setting

In this section, we describe how we applied our approach to an industrial product line at Hitachi. The domain of the product line is a kind of embedded control system that consists of sensors and actuators to control a mechanical system. This system was designed as a software product line and products were generated with selecting large number of variation points that included 123 optional features. When we conducted the case study, the organization derived 8,513 products from the product line infrastructure. We evaluated our method by apply it to the selections of optional features for the products.

Our approach for mining a database requires effective tool support because of the large number of feature selections that need to be processed. For association rule learning, we adopted *R* [8] which provides an interface for association rule computing.

In this case study, we defined the threshold measures *supp*, *conf*, and *lift* as 1%, 95%, and 2.0, and identified 31 association rules as recommendations of new feature constraints.

3.2 Evaluation

We evaluated the recommendations to see if they are real new feature constraints of the product line. In this case study, we compared the recommendations with the specifications of the product line and interviewed expert engineers. Table 3 shows the evaluation results of the case study.

First, we found that three of the recommendations were already specified as feature constraints in the current specification.

Then, we examined the undocumented recommendations. The results were that the recommendations include seven spurious relationships. Rule 4 in Table 2 is an example of a spurious relationship. This rule $\{\neg\text{feature } 4 \Rightarrow \text{feature } 2\}$ is created by an intervening feature $\{\neg\text{feature } 4 \Rightarrow \text{feature } 3 \Rightarrow \text{feature } 2\}$ (Rule 1 and 2).

Finally, we asked expert engineers if the rest of recommendations are real new feature constraints in the product line. The engineers examined 21 recommendations and accepted them as new feature constraints.

Table 3. Evaluation results

Classifications	# rules	%
Already identified in spec.	3	10%
Spurious relationships	7	23%
New feature constraints	21	68%
Total	**31**	**100%**

4 Related Work

Savolainen et al. [9] described a way for observing the optional features by focusing on their usage pattern and evolving their default values. Savolainen et al.'s method reduces the configuration effort of optional features, but their approach does not consider the combinations of features. Our approach extends their concept of the usage pattern observation to the relationship between features, i.e., feature constraints.

Loesch and Ploedereder [10] provided a way to optimize variability in a product line. They reconstructed the variability model based on the concept lattice model, so there is a gap to migrate into the feature-oriented approach that is widely used. However, our approach can complement the feature-oriented approach with the association rules of the features in the field.

5 Conclusion

Identifying new and changing feature constraints is a key activity for product line evolution. This paper described a novel approach enabling us to recommend

new feature constraints by applying association rule learning to feature selections made in the past. Our approach can be a helpful tool for identifying new feature constraints.

Although statistical analysis is a reactive approach to trace the evolution of product lines, knowing about patterns that appear frequently in feature selections may provide great insight to organizations. Analyzing and refactoring core assets, especially for large-scale product lines, becomes exponentially more difficult. Therefore, we will extend the method in this paper and take statistical methods as a key approach for continuous product line evolution.

References

1. Krueger, C.W.: The 3-tiered methodology: Pragmatic insights from new generation software product lines. In: SPLC 2007: Proceedings of the 11th International Software Product Line Conference, pp. 97–106 (2007)
2. Bosch, J.: Software product families in Nokia. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 2–6. Springer, Heidelberg (2005)
3. Reiser, M.O., Weber, M.: Managing highly complex product families with multi-level feature trees. In: RE 2006: Proceedings of the 14th IEEE International Requirements Engineering Conference, pp. 146–155 (2006)
4. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
5. Weiss, D.M., Li, J.J., Slye, H., Dinh-Trong, T., Sun, H.: Decision-model-based code generation for sple. In: Proceedings of the 12th International Software Product Line Conference, SPLC 2008, pp. 129–138 (2008)
6. Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J.H., Pohl, K.: Variability issues in software product lines. In: van der Linden, F.J. (ed.) PFE 2002. LNCS, vol. 2290, pp. 13–21. Springer, Heidelberg (2002)
7. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. In: SIGMOD 1993: Proceedings of the, ACM SIGMOD International Conference on Management of Data, pp. 207–216 (1993)
8. The R Foundation: The R project for statistical computing, http://www.r-project.org/ (Visited on Febraury 12, 2010)
9. Savolainen, J., Bosch, J., Kuusela, J., Männistö, T.: Default values for improved product line management. In: SPLC 2009: Proceedings of the 13th International Software Product Line Conference, pp. 51–60 (2009)
10. Loesch, F., Ploedereder, E.: Optimization of variability in software product lines. In: SPLC 2007: Proceedings of the 11th International Software Product Line Conference, pp. 151–162 (2007)

Software Product Line Engineering for Long-Lived, Sustainable Systems

Robyn Lutz[1,2], David Weiss[1], Sandeep Krishnan[1], and Jingwei Yang[1]

[1] Department of Computer Science, Iowa State University
[2] Jet Propulsion Lab/Caltech
{rlutz,weiss,sandeepk,jwyang@cs.iastate.edu}

Abstract. The design and operation of long-lived, sustainable systems (LSS) are hampered by limited support for change over time and limited preservation of system knowledge. The solution we propose is to adopt software product-line engineering (SPLE) techniques for use in single, critical systems with requirements for sustainability. We describe how four categories of change in a LSS can be usefully handled as variabilities in a software product line. We illustrate our argument with examples of changes from the Voyager spacecraft.

Keywords: software product line, sustainable system, long-lived system, variability, commonality/variability analysis.

1 Introduction

Sustainable: "meets the needs of the present without compromising the ability of future generations to meet their own needs"

- UN Brundtland Report, on sustainable development [1].

Our society is becoming increasingly dependent on software-intensive sustainable systems. Examples include embedded medical devices, web-based archives, interplanetary spacecraft, power grid monitors, telecommunication switches, and sensor networks. Future examples include nuclear power plants, health databases, and global networks of solar arrays, perhaps in orbit. Many such systems are safety critical, with varying degrees of autonomy. They typically evolve over long periods of time in response to changed needs, new technologies, and failed components.

We consider a sustainable system to be one that has the following attributes.

- It has an extended lifetime,
- It makes efficient use of resources to achieve its goal.
- It maintains its capabilities despite obstacles and failures.
- It is adaptable, so as to accommodate change, and is expected to evolve with changes in technology and requirements.

More broadly, a sustainable system is forward-looking and is structured so as to guide future decisions. The goal of evolving over time to meet changes in technology and

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 430–434, 2010.

requirements, distinguishes sustainable systems from legacy systems. Accordingly we use the term long-lived, sustainable systems, or LSS, for them.

In this paper our perspective is the preservation of system knowledge over time in the service of handling change (both anticipated and unanticipated) in LSS. While the preservation of knowledge and change handling are not unique to LSS, extended LSS lifetimes exacerbate the problems. LSS have a longer period of operations over which both planned and unplanned change can occur. Their long operational periods are accompanied by considerable personnel turnover, resulting in knowledge loss that complicates operations and adaptive maintenance. Historically, these inadequacies have jeopardized LSS [2]. Their design and maintenance is challenged by the need to envision, plan for, and handle on-going change, and to preserve and pass on the knowledge needed to do so.

The problem, then, is *how to better design and operate a LSS to preserve system knowledge and to support needed changes over time*. The solution we propose is to adopt software product-line engineering (SPLE) techniques for use in single LSS, an adoption that we believe is natural to both. SPLE provides a process framework to identify, document, and make decisions regarding alternatives now and in the future, taking into consideration their risks, dependencies and consequences, both in cost and value. It focuses on sustaining artifacts and domain knowledge over a long haul.

Change can be usefully treated as variability, and SPLE handles variability well. To illustrate this we discuss examples of anticipated and unanticipated changes from the twin Voyager spacecraft, launched in 1977 and still actively collecting science data. We show how Voyager "did it right" in planning and organizing for change, and in maintaining system knowledge.

We suggest that the lessons to be learned from Voyager regarding the design and operation of a LSS system are not just consistent with software product line engineering but are, in fact, most readily transferable to other LSS in the context of SPLE techniques. SPLE describes how a set of similar systems—a software product family—develops over time. We treat an evolving LSS as if it were a set of similar systems that developed over time. (In fact, PLE also evolves over space, that is, there may be several products in a product line that are produced and maintained concurrently, so our problem is simpler.) This paper thus proposes *to apply SPLE techniques to single systems, where those single systems must be long-lived and sustainable,* and presents, in the context of Voyager, the advantages of doing so.

2 LSS Example: The Voyager Spacecraft

The two Voyager spacecraft, launched in 1977 and now the farthest human-made objects from Earth, are among the best-known LSS. The spacecraft continue to return truly invaluable data as they approach the heliopause. For example, Voyager 2 recently discovered a strong magnetic field that holds the interstellar cloud together [http://voyager.jpl.nasa.gov/]. The Voyagers are expected to continue to communicate until loss of power and fuel mutes them around 2020. The spacecraft have efficiently used their early-1970's era resources to adapt to a changing set of ambitious scientific goals. The spacecraft software has also been repeatedly changed to handle failed components and reduced power. Voyager did not explicitly use software product line

engineering. However, viewed in retrospect, Voyager exemplified the SPLE process of carefully identifying possible variations that might be needed in the future, of designing a modularized architecture that would allow those anticipated changes to be made, and of specifying the constraints that would guide the decisions to be made. Voyager also demonstrated that even unanticipated change is made easier when a serious effort has been made to design for possible future changes.

3 SPLE for LSS Change Management

The software product line engineering FAST process used here, identifies, distinguishes, and documents what is assumed to stay the same (across systems and time) and what may change [3, 4]. It relies on three artifacts: (1) a commonality/variability analysis that formally specifies the allowable range of values for each variability, the constraints among the choices of value for the variabilities, and the binding time for each variability; (2) a modularized architecture with a mapping between modules and the commonality/variability specifications described above; and (3) a specification, called a Decision Model, of the partially-ordered sequence of choices that must be made to build a new product, subject to the constraints and binding times specified earlier.

Some required behavior must be invariant for a LSS to succeed. For example, a software requirement that has to be satisfied throughout the lifetime of the Voyager spacecraft is that it shall be able to communicate with Earth and automatically detect and respond to a loss of uplink from Earth. Similarly, a LSS is built and operated on certain assumptions regarding those things that will not change (e.g., in the environment). Such assumptions can usefully be modeled as commonalities. Note also that if the assumptions later become false, we have a way in FAST to document both what the change is and why the change occurred, preserving knowledge and providing guidance to later generations of maintainers, as suggested in [5].

We next describe how the handling of both anticipated and unanticipated change can be improved by the use of software product line engineering techniques.

3.1 Anticipated Changes

We can anticipate some changes that likely will be made during the lifetime of a LSS. Several standard techniques assist in this identification: investigation by domain experts, experience with similar systems, goal/obstacle analysis, defect patterns in similar systems, and analysis of previous changes. On spacecraft, we know that if hardware breaks, the software will often have to be updated to take on the required capability previously allocated to hardware [2]. Similarly, we know that as different mission phases are reached (e.g., launch, interplanetary cruise, planetary) the software will need to be updated. In LSS, many of these changes will be made to handle failure or degradation of hardware components.

Such anticipated changes can usefully be modeled as product line variabilities in the commonality/variability analysis. The FAST process documents the envisioned ranges of optional and alternative requirements and parameters. Making an anticipated change after launch then becomes analogous to taking a different path through the Decision Model. In so doing, you use the Decision Model to check that

the impact of the change is acceptable, based on the constraints between the proposed alternative and the choices implemented earlier to produce the existing product. This provides some assurance that the software architecture can accommodate the change. Checking the constraints can often be partially or fully automated.

Modifiability, or "changeability", is a defining attribute for a LSS. In [6], the quality attributes of modifiability are organized into four categories: (1) extensibility (changing capabilities, adding new functionality, repairing bugs); (2) deleting existing capabilities; (3) portability (adapting to new operating environments); and (4) restructuring (modularizing, optimizing, or creating reusable components).

We describe critical changes that occurred on the Voyagers during operations for the first three of these categories. Because of page limits, we exclude the fourth category here, but note that several subsequent spacecraft re-used Voyager hardware and software. We interleave examples from the two spacecraft, despite some small differences between them. For each change, we show how it fits into a SPLE context.

Anticipated extensibility: At launch, the Voyagers' authorized flight plan included only Jupiter and Saturn, primarily for budget reasons. However, the spacecraft had been designed for extension to take advantage of the fact that Jupiter, Saturn, Uranus and Neptune were aligned, something that happens once every 175 years. When the flight was extended for the "Grand Tour" to all four planets, the architectural design was in place to allow this. Note that the cost of designing for this extensibility was far outweighed by the value of the scientific knowledge gained from it.

Anticipated deletion: It was known that each instrument drawing significant power would have to be turned off at some point, as the onboard battery capacity decreased. For example, the cameras were turned off in 1990 after the last planetary encounter. Weighing the tradeoffs and deciding when in the mission to turn each instrument off was a complicated task, but made possible by anticipating its need.

Anticipated portability: A new algorithm was required to obtain images at Neptune. Without it, the low sunlight levels, combined with the torque imparted when the tape recorder was turned on and off, would have caused images to be smeared. The new feature automatically fired the attitude jets to compensate for the spacecraft torque at the longer exposure rate.

3.2 Unanticipated Changes

Unanticipated change is a problem for any system. SPLE provides a structure for dealing with unanticipated change as well as anticipated change. In particular, it allows one to reason about the effects, dependencies, and risks of a proposed change.

Unanticipated extensibility: In 1987, a new science opportunity appeared. A supernova occurred that Voyager could observe. Software commands were thus designed and sent to the UV spectrometer to capture data from the stellar explosion, taking advantage of a design that anticipated the need to reprogram the spacecraft.

Unanticipated deletion: Slewing rates for the scan platform (containing the cameras, etc.) were unexpectedly restricted by a new project policy for the planetary encounters of Uranus and Neptune. The change was in response to an earlier incident where the platform jammed, likely induced by a period of heavy usage.

Unanticipated portability: Soon after launch, Voyager 2's primary receiver failed and its backup receiver was reduced to "hearing" in a very narrow, changing frequency band. To compensate, a new ramping algorithm was quickly designed and implemented, so that prior to sending any software commands to the spacecraft, ground operations could tune the transmission to the receiver's current state.

SPLE excels at identifying what needs to be known and storing it so that the new customer (here, the spacecraft team) need only be concerned with the information preserved in the structures, rather than having to learn all the underpinnings of the system. Because the system is specified and designed for change, new personnel know where to look to understand the system and the implications of change.

4 Conclusion

Our hope is that use of SPLE techniques will make it easier to make changes to LSS where the value/cost ratio is high, as it is on spacecraft and other critical or one-of-a-kind systems. It remains to test this hypothesis, perhaps as a shadow effort with an on-going LSS project. In particular, we think that these SPLE techniques will be easy to use and fit in readily with the way sustainable projects work.

The benefits that we anticipate may accrue from the use of SPLE for LSS include:

- Improved preservation of project knowledge over extended lifetimes, leading to lower cost to maintain.
- Better capture of assumptions (commonalities) and dependencies among choices (variabilities) that can help reduce risk of change.
- Increased emphasis on investigating and ranking possible changes and on verifying architectural support for modifiability.
- Viewing potential changes as options that, when exercised, can bring high value, and so merit investment to preserve the needed information.

Acknowledgments. This work was supported by grants 0541163 and 0916275 from the National Science Foundation.

References

1. Our Common Future, Report of the World Commission on Environment and Development. Oxford University Press, Oxford (1987)
2. Lutz, R.R., Mikulski, I.C.: Empirical Analysis of Safety Critical Anomalies during Operation. IEEE Trans. Software Engineering 30(3), 172–180 (2004)
3. Weiss, D.M., Lai, C.T.R.: Software Product-Line Engineering, A Family-Based Software Development Process. Addison-Wesley, Reading (1999)
4. Weiss, D.M., Li, J.J., Slye, H., Dinh-Trong, T., Sun, H.: Decision-Model-Based Code Generation for SPLE. In: SPLC 2008, pp. 129–138 (2008)
5. Parnas, D.L., Clements, P.C.: A rational design process: How and why to fake it. IEEE Trans. on Software Engineering 12(2), 251–257 (1986)
6. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, Reading (1998)

An Approach to Efficient Product Configuration in Software Product Lines

Yuqing Lin, Huilin Ye, and Jianmin Tang*

School of Electrical Engineering and Computer Science
The University of Newcastle, NSW2308, Australia
{yuqing.lin,huilin.ye,jianmin.tang}@newcastle.edu.au

Abstract. Feature modeling has been widely used in software product line engineering to represent commonality and variabilities among products in a product family. When developing a new software product belonging to a product line, a feature model representing the product line will be used to configure products. The product configuration process is a decision making process, various kinds of constraints and complex relationships among configurable features make the decision making a time consuming and error prone task. In this paper, we present an approach which will improve the efficiency and quality of product configuration.

1 Introduction

Software product line is a collection of software products that share common characteristics as a family in a specific application domain. In addition to the commonalities shared by all the products in a SPL, individual products might vary significantly. The variabilities among the products in a SPL must be appropriately represented and managed. Feature oriented modeling approaches have been widely used in software product line engineering for this purpose. Features are prominent and distinctive system requirements or characteristics that are visible to various stakeholder in a product line [2]. A feature model specifies the features, their relationships, and the constraints of feature selection for product configuration in software product lines. There are a lot of work been done on the automated analysis of the feature model, for example, identifying the void feature model, validating software product etc. for more detail, see [3].

In this paper, we will be focusing on the product configuration using the feature model. Product configuration is a process of selecting features for developing a product in a SPL. A product in a SPL is defined by a unique combination of legally selected features in the SPL. A feature model is usually represented as a tree in which the variabilities of features are represented as variation points. A variation point consists of a parent feature, a group of variable child features, called variants, and a multiplicity specifying the minimum and maximum number of variants that can be selected from the variation point when configuring

* This work is supported by Australian Research Council Discovery Project (DP 0772799).

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 435–439, 2010.

a product. The selection of variants at a variation point is not only constrained by the multiplicity but also by the dependencies between the variants at this variation point and the variants at other variation points. The following two dependencies have been identified by Kang et al [2].

1. Requires: If a feature requires, or uses, another feature to fulfil its task, there is a Requires relationship between the two features.
2. Excludes: If a feature conflicts with another feature, they cannot be chosen for the same product configuration, i.e. they mutually exclude each other. There is a bi-directional Excludes relationship between the two features.

When configuring a product we usually need to go through the whole feature tree and make a configuration decision at each variation point to select variant(s) of the variation point. Various issues in the product configuration have been discusses, for example, in [4], authors have proposed a framework to use reasoning algorithms to provide automated support for Product Configuration, especially for Collaborative Product Configuration. Other approaches treat the configuration process as a constraint satisfaction problem (CSP) [7] [5]. These approaches provide support or solutions for the decision making of the configuration process. In this paper, we will be optimizing the configuration process to make the configuration process more efficient.

2 Configuration Coverage

Before we present the proposed approach the following measures are defined for the approach.

– Positive coverage and negative coverage of variant:

When a variant v at a certain VP is selected for a product configuration, positive coverage of v, represented as PC(v), is a set of variable features that will be automatically included in or excluded from the product based on their dependencies with v. When a variant v at a certain VP is excluded for a product configuration negative coverage of v, represented as NC(v), is a set of variable features that will be automatically included in or excluded from the product based on their dependencies with v.

– Configuration Coverage (CC) of a valid selection at a variation point:

For all the variants associated with a variation point, we call a subset of variants a valid selection if it obeys the multiplicity of the variation point. The complement of a valid selection is the set of variants that are not included in the selection at the variation point. When a certain valid selection has been made at a variation point the configuration coverage of the variation point is the union of all the positive coverages of the variants in the valid selection and all the negative coverages of the variants in the complement of the valid selection. Below is an example to illustrate how to calculate the above defined measures

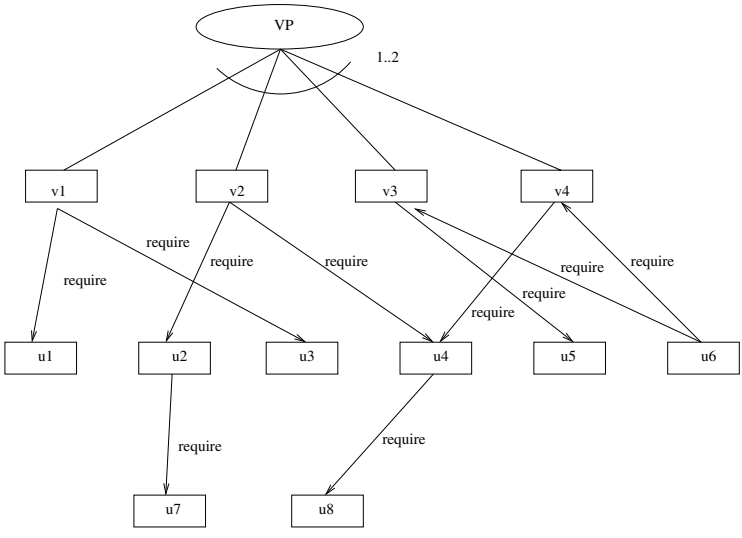


Fig. 1. A variation point VP and its variants

In the Fig 1 we show a fraction of a feature model. There are 4 variants $v1$, $v2$, $v3$ and $v4$ associated with the variation point VP. The multiplicity specifies that only up to 2 variants can be selected in a product configuration. Based on the dependency relationships shown in teh same figure, we know that:$PC(v1) = \{u1, u3\}$, $NC(v1) = \emptyset$, $PC(v2) = \{u2, u4, u7, u8\}$, $NC(v2) = \emptyset$, $PC(v3) = \{u5\}$, $NC(v3) = \{u6\}$, $PC(v4) = \{u4, u8\}$, and $NC(v4) = \{u6\}$.

All possible selections at the VP are listed below,

$v1$,$v2$, $v3$, $v4$, $v1 \cup v2$, $v_1 \cup v3$, $v_1 \cup v_4$, $v2 \cup v3$, $v2 \cup v4$, $v3 \cup v4$

The configuration coverage (CC) of each valid selection is listed as following:

$$CC(v1) = PC(v1) \cup NC(v2) \cup NC(v3) \cup NC(v4) = \{u1, u3, u6\},$$

$$CC(v2) = \{u2, u4, u7, u8, u6\},\ CC(v3) = \{u5, u6\},\ CC(v4) = \{u4, u8, u6\},$$

$$CC(v1 \cup v2) = PC(v1) \cup PC(v2) \cup NC(v3) \cup NC(v4) = \{u1, u2, u3, u4, u7, u8, u6\},$$

$$CC(v1 \cup v3) = \{u1, u3, u5, u6\},\ CC(v1 \cup v4) = \{u1, u3, u4, u6\},$$

$$CC(v2 \cup v3) = \{u2, u4, u7, u8, u5, u6\},\ CC(v2 \cup v4) = \{u2, u4, u7, u8, u6\},$$

$$CC(v3 \cup v4) = \{u5, u6\}.$$

The maximum configuration coverage from the above set of CCs is the CC{v1, v2} when $v1$ and $v2$ are selected at this variation point. We call this maximum configuration coverage $MAXCC$. $MAXCC$ of a variation point is the maximum CC over all possible selection of variants at the variation point. The bigger the $MAXCC$, the (potentially) more variable features will be included/excluded once a decision made at the variation point, therefore, the more important is the variation point to be considered earlier in the configuration process.

3 The Proposed Approach and Case Study

Our proposed approach will make use of the measurement $MAXCC$, using which we can identify a small set of variation points from a feature model, the $MAXCC$ of this set of variation points will cover all the variant selections in the whole feature model. The smallest is the set, the less decisions are to be made in the product configuration process, implies that more efficient and less error-prone of the configuration.

To identify this set of variation points, we could use a simple greedy approach, i.e. selecting the variation points with the biggest coverage until the union of the coverage covers the feature model. A more precise approach is to model the problem as the minimum vertex cover problem and use some approximation algorithm to solve the problem. To do that, we will first transfer the feature model into a directed graphs, The transformation is quite straightforward, every variable feature in the feature model become a vertex in the resulting graph and the dependencies between two variable features become the arcs in the resulting graph. Once we model the feature model into a direct graph, we can then take our problem as a discrete optimization problem. In graph theory, a "vertex-cover" of a directed graph (digraph) is a set of vertices such that each arc of the digraph is incident to at least one vertex of the set. A minimum vertex-cover is a vertex-cover of the smallest size. The problem of finding a minimum vertex-cover is a classical optimization problem in computer science. In our experiment, we have used the HSAGA algorithm [6] which is efficient and very often can produce good solutions.

A case study based on a Library Software Product Line demonstrating how the proposed approach works has been conducted. A feature model was established for the product line. It contains 23 variation points, 35 variants, and complex dependencies between the variants. We applied the proposed approach to the feature model and identified a sequence of variation points containing only 10 variation points but cover the whole variant selections of the Feature Model. Instead of making decisions at 23 variation points we can now only make selection at the 10 identified variation points, which will increase the configuration efficiency and reduce the chances of making wrong configuration decisions.

4 Conclusions and Future Works

In the product configuration, software engineers should start with this set of variation points when configuring a product from the feature model. This set of variation points represent the key decisions for configuring a member product. Focusing on this set of variation points will reduce the configuration effort for software engineers as the decisions made on these variation points will imply the decisions on other variation points in the feature model. Another advantage is that the software engineers are unlikely to selecting conflicting variants since the dependency constraints in a feature model have been encoded in CCs. In this approach, we only considered the Requires and Exclude relationships, there are

other complex dependencies exist in the feature model. In the future, we would like to develop a more flexible model so we can deal with more complex dependency relationships. A formal evaluation on the performance of the proposed approach will also be done in the near future.

References

1. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multilevel configuration of feature models. Software Process: Improvement and Practice 10(2), 143–169 (2005)
2. Lee, K., Kang, K., Lee, J.: Concepts and Guidelines of Feature Modelling for Product Line Software Engineering. In: Gacek, C. (ed.) ICSR 2002. LNCS, vol. 2319, pp. 62–67. Springer, Heidelberg (2002) ISBN: 978-3-540-46020-9
3. Benavides, D., Segura, S.: A. Cortes Automated Analysis of Feature Models 20 Years Later: A Literature Review. Information Systems (in press)
4. Mendonca, M., Cowan, D.: Decision-making coordination and efficient reasoning techniques for feature-based configuration. Science of Computer Programming 75(5), 311–332 (2010)
5. Mendonca, M., Wasowski, A., Czarnecki, K., Cowan, D.: Efficient compilation techniques for large scale feature models. In: Proceedings of the 7th international conference on Generative programming and component engineering, pp. 13–22 (2008)
6. Tang, J., Miller, M., Lin, Y.: HSAGA and its application for the construction of near-Moore digraphs. Journal of Discrete Algorithms 6(1), 73–84 (2008)
7. Felfernig, A., Friedrich, G., Jannach, D., Zanker, M.: Towards distributed configuration. In: Baader, F., Brewka, G., Eiter, T. (eds.) KI 2001. LNCS (LNAI), vol. 2174, pp. 198–212. Springer, Heidelberg (2001)

A Hybrid Approach to Feature-Oriented Programming in XVCL

Hongyu Zhang[1] and Stan Jarzabek[2]

[1] School of Software, Tsinghua University, Beijing 100084, China
hongyu@tsinghua.edu.cn
[2] School of Computing, National University of Singapore, Singapore 117543
stan@comp.nus.edu.sg

Abstract. Feature-Oriented Programming (FOP) is a programming paradigm for developing programs by composing features. It is especially useful for software product line development, as each product line member implements some combinations of features. FOP attempts to modularize features and to enable their flexible composition into programs. Recent studies have shown that it is not practical to modularize and then compose features that have fine-grained impact on base programs. In this paper, we present a hybrid approach to feature modularization/composition problem. We modularize only separable features that can be well contained in dedicated files. We handle inseparable features by annotating base programs using preprocessing-like directives. We show how the hybrid approach can be achieved in XVCL, a generative technique to manage variabilities in software product lines.

1 Introduction

Feature-Oriented Programming (FOP) is a programming paradigm for developing programs by composing features [2, 11]. FOP extends the principle of separation of concerns to features. It attempts to modularize feature implementation and provides a mechanism to compose features into a base program in required, legal combinations. The ability to compose features in flexible way is particularly useful for software product line development, where we need to handle a family of similar products characterized by common features, but with each product implementing some unique features.

Researchers have experimented with techniques that can realize the concept of FOP. AHEAD [2] is among the mature techniques. Some researchers also suggested Aspect-Oriented Programming (AOP) as a possible technique to realize FOP [1, 10]. Both AHEAD and AOP can handle the *separable features*, which can be well modularized in dedicated files and can be composed into base programs at a relatively small number of variation points. Recently Kastner and Apel found that AHEAD and AOP have limitations in handling features that have fine-granular impact on base programs at many variation points [7, 8]. These limitations could lead to the excessive use of inheritance and hook-methods, causing difficulties in program understanding and maintenance. Such *inseparable features* create a major challenge for FOP.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 440–445, 2010.

In this paper, we present a hybrid approach to feature modularization/composition problem. We modularize only separable features and handle inseparable features by annotating programs using preprocessing-like directives. We show a realization of the proposed hybrid approach in XVCL (XML-based Variant Configuration Langue) [4, 5, 14], which is a generative technique to manage variabilities in software product lines. Although XVCL has been applied in many product line and software maintenance studies, its usage in FOP was not well explored before.

Unlike other approaches such as AHEAD or AOP, our approach does not force all the features to be separated and modularized. Our approach is both annotative and compositional. For features that can be well separated, we modularize their implementations by placing their code in separate modules. When clean feature modularization becomes problematic, we directly annotate the base programs with necessary feature code. By doing so, we avoid the limitations of current FOP techniques (such as the excessive use of inheritance and hook-methods), and improve the maintainability of the feature programs. We allow developers to choose the method for handling different features, thus achieving higher degree of flexibility in programming practices.

2 A Hybrid Approach to FOP

Feature-Oriented Programming (FOP) is based on the principle of advanced separation of concerns [13]. Features are often treated as a form of concerns that can characterize software products. In FOP, each module (or each layer of modules) localizes the code implementing a feature. If we could find a way to clearly separate all the features using FOP, no doubt our software engineering problems would be much less than they are today. However, some of the features are so tightly coupled with other features or with the base programs that their physical separation becomes difficult. We thus classify the features into two types:

Separable features: Features that can be easily separated from the base programs. The impact of these features can be well modularized in dedicated files and can be composed into base programs at a relatively small number of variation points.

Inseparable features: Features that are difficult to be separated from the base programs. Examples of such features include:

- Features that require fine-grained changes. The examples of fine-grained changes include changes in method signature, changes in the middle of a method, and extensions of expressions. To handle such fine-grained changes, the existing modularization and composition mechanisms often lead to complicated and unmaintainable implementations.
- Features that are inherent properties of base programs. These features are integral parts of the descriptions of concepts. They cannot be separated from the base programs without hampering the program integrity and understandability. For example, for a Buffer class, its data type (such as Int, Char, Byte, Double, etc) is an integral property of the class. Therefore it is better to implement this data type feature as generics, instead of separating it into independent modules.

- Features that have intensive interactions with other features. Some features heavily crosscut other features. One example of such feature is exception handling. Separating the exception handling code from the base program is shown to be difficult. Performance and security features are other examples. Performance/Security has pervasive impact on many design decisions. While we can conceive and express performance/security concerns conceptually (e.g., by documenting design decisions that have to do with performance/security), "physical" modularization of these features may not be feasible.

To handle separable features, we can modularize their impact into dedicated modules, and then compose these modules with the base programs. This process is the same as what current FOP approaches do. The modularization of feature-specific code helps programmers understand and maintain features. To handle inseparable features, we can directly annotate the base programs to incorporate the impact of these features. The basic concept is close to the concept of C/C++ preprocessing directives. We can thus understand the base programs and the inseparable features as a whole. In this way, we avoid the limitations of current FOP approaches (such as the excessive use of inheritance and hook-methods).

3 A Realization of the Hybrid FOP in XVCL

3.1 An Overview of XVCL

XVCL (XML-based Variant Configuration Language) is a variation mechanism for managing variability in software product lines based on generative techniques [3]. XVCL has been applied to support product lines in lab studies and industrial projects [4, 5, 14].

In XVCL, generic and adaptable programs are called x-frames. X-frame body is written in the base language, which could be a programming language such as Java or PHP. An x-frame is also marked-up by XVCL commands (in the form of XML tags), which enables the composition and adaptation of the x-frame. Typical XVCL commands include the <adapt> command that composes two x-frames, the <select> and <ifdef> commands that allow one to select pre-defined options based on certain conditions, the <break> command that marks breakpoints (slots) where additional code can be inserted, and the <insert> command that inserts additional code into a specified slot. To derive a customized program from x-frames, one needs to design a Specification x-frame (SPC), which records specific product configurations according to the user requirements. Given SPC, the tool *XVCL processor* performs composition and adaptation by traversing x-frames along <adapt> chains, executing XVCL commands embedded in visited x-frames, and constructing concrete programs.

XVCL is a software reuse technique based on the idea of "composition with adaptation". Like macros, x-frames contain code fragments along with adaptation commands. Unlike macros, XVCL has unique capabilities that enhance reuse, such as scoping and overriding rules for meta-variables, insertions at breakpoints and while loops that facilitate generating custom component instances from a generic x-frame. Unlike OO inheritance, C++ template and Aspect-Oriented Programming (AOP)

techniques, XVCL can handle variants at any granularity level. One can explicitly mark the variation points in a program and specify required adaptations.

More details about XVCL can be found at the XVCL homepage[1]. Although XVCL has been applied to many product line and software maintenance studies, its usage in FOP was not well explored before.

3.2 Implementing FOP Using XVCL

We use XVCL to realize the proposed hybrid approach to FOP. To implement inseparable features, we use XVCL commands (such as <select> and <ifdef>) to directly mark the variation points in base programs at which customization will occur. These commands will select necessary code to be included into the generated program based on the values of meta-variables that are set in the specification x-frame (SPC).

We place the implementation of separable features in dedicated x-frames. To support feature modularization, we use the XVCL <select> commands to encapsulate all necessary code fragments for implementing a feature. We also use the XVCL <break> commands to specify the slots in the base programs at which possible extensions could be made. During program composition, we select suitable code fragments in the feature x-frame and insert them into the slots defined in the base programs.

In our XVCL-based hybrid approach to FOP, we develop two types of x-frames:

- x-frames that implement the base programs. The base programs can be developed in the conventional object-oriented manner and can be annotated with XVCL commands to accommodate inseparable features.
- x-frames that implement separable features.

These two types of x-frames can be composed together to form a complete, compilable program. A valid combination of features for a specific system is described in an SPC. The XVCL processor performs the composition of base classes and features according to the instructions given in the SPC, and generates a specific system that implements the required features.

3.3 Experiments

We have evaluated our approach through the development of the Graph Product Line (GPL). GPL has been proposed by Lopez-Herrejon and Batory [9] as a standard problem for evaluating software product line technologies. It is a family of classic graph applications. In GPL, we identify *Directed/Undirected* and *Weighted/Unweighted* features as inseparable features. These features are inherent properties of the base programs. Their code is tightly coupled with the base program at the fine-granular level and cannot be easily isolated. Therefore we use the <ifdef> commands to annotate the base program with the optional feature code. We modularize other features (such as the *Cycle* feature) in dedicate x-frames. By separating these features from the base

[1] `http://xvcl.comp.nus.edu.sg`

programs, we enhance the readability of the base programs. Given a set of features, a specific GPL product can be generated from the x-frames.

To further evaluate our method, we re-engineered the Berkeley DB system into hybrid FOP representations. The Berkeley DB case study was also used by Kastner et al. [7], who refactored the system into features using AspectJ. In our study, we reengineered 23 Berkeley DB features into x-frames. Features such as *EvictorDaemon* are implemented as inseparable features, features such as *FSync* are implemented as separable features.

3.4 Tool Support

To facilitate XVCL-based development, especially for a large and complex product line, we have also developed a set of tools including a development workbench (an IDE for editing x-frames), a feature configuration checker (for checking the validity of a selected feature combination), and a feature query tool (for analyzing the impact of a feature on x-frames). These tools support the development of x-frames, their reuse and evolution, mitigating potential problems of understandability and scalability of complex x-frames. More details about these tools can be found at [6, 12].

4 Conclusions

Feature-Oriented Programming (FOP) is a programming paradigm for developing programs by composing features. We classify features into two types: 1) inseparable features that affect base programs at fine-granular level and cannot be easily separated from base programs, and 2) separable features that can be easily separated from base programs and contained in dedicated modules. Recent studies show that current FOP techniques based on advanced separation of concerns principle, such as AHEAD and AOP, have limitations in implementing inseparable features.

In this paper, we have presented a hybrid approach to feature modularization/ composition problem. We modularize only separable features and handle inseparable features by annotating programs using preprocessing-like directives. We have presented a realization of the above approach in XVCL, a generative technique to manage features in software product lines.

In future, we plan to further investigate new tools that can facilitate XVCL-based FOP development. One trade-off in our approach is that we cannot guarantee correctness of programs generated from meta-programs (x-frames). We will address this issue in future work. We also plan to carry out a larger-scale evaluation of the proposed hybrid approach.

Acknowledgments. We would like to thank Roberto E. Lopez-Herrejon for providing us the source code of their implementation of GPL in AHEAD. This research is supported by the Chinese NSF grant 60703060, the NUS research grant RP-252-000-336-112, and the MOE Key Laboratory of High Confidence Software Technologies at Peking University.

References

1. Apel, S., Leich, T., Saake, G.: Aspectual Feature Modules. IEEE Trans. Software Eng. 34(2), 162–180 (2008)
2. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Eng. 30(6), 355–371 (2004)
3. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, MA (2000)
4. Jarzabek, S.: Effective Software Maintenance and Evolution: Reuse-based Approach. CRC Press, Taylor & Francis (2007)
5. Jarzabek, S., Zhang, H.: XML-based method and tool for handling variant requirements in domain models. In: Proc. Int'l. Symp. on Requirements Engineering (RE 2001), Toronto, Canada (2001)
6. Jarzabek, S., Zhang, H., Lee, Y., Xue, Y., Shaikh, N.: Increasing Usability of Preprocessing for Feature Management in Product Lines with Queries. In: Proc. ICSE 2009, Vancouver, Canada, May 2009, pp. 215–218 (2009)
7. Kästner, C., Apel, S., Batory, D.: A Case Study Implementing Features Using AspectJ. In: Proc. Int. Software Product Line Conference (SPLC 2007), Kyoto, Japan, September 2007, pp. 223–232 (2007)
8. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in Software Product Lines. In: Proc. ICSE 2008, Leipzig, Germany, May 2008, pp. 311–320 (2008)
9. Lopez-Herrejon, R.E., Batory, D.: A standard problem for evaluating product-line methodologies. In: Bosch, J. (ed.) GCSE 2001. LNCS, vol. 2186, pp. 10–24. Springer, Heidelberg (2001)
10. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. In: Proc. SIGSOFT FSE 2004, Newport Beach, CA, pp. 127–136 (2004)
11. Prehofer, C.: Feature-oriented programming: A new way of object composition. Concurrency and Computation: Practice and Experience 13(6), 465–501 (2001)
12. Sun, J., Zhang, H., Li, Y., Wang, H.: Formal Semantics and Verification for Feature Modeling. In: Proc. 10th Int. Conf. on Engineering of Complex Computer Systems (ICECCS 2005), Shanghai, June 2005, pp. 303–312 (2005)
13. Tarr, P., Ossher, H., Harrison, W., Sutton Jr., S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proc. ICSE 1999, Los Angeles, CA, USA (May 1999)
14. Zhang, H., Jarzabek, S.: XVCL: A Mechanism for Handling Variants in Software Product Lines. Science of Computer Programming 53(3), 381–407 (2004)

An Approach for Developing Component-Based Groupware Product Lines Using the Groupware Workbench

Bruno Gadelha[1], Elder Cirilo[1], Marco Aurélio Gerosa[2], Alberto Castro Jr.[3], Hugo Fuks[1], and Carlos J.P. Lucena[1]

[1] Department of Informatics, Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
R.M.S Vicente, 225, Gávea, Rio de Janeiro - RJ, Brazil, 22453-900
{bgadelha,ecirilo,hugo,lucena}@inf.puc-rio.br
[2] Computer Science Department, University of São Paulo (USP)
R. Matão, 1010, São Paulo 05508-090, Brazil
gerosa@ime.usp.br
[3] Department of Computer Science, Federal University of Amazonas (UFAM)
Av. Gal. R.O.J. Ramos, 3000, Manaus, Brazil
alberto@ufam.edu.br

Abstract. Groupware are computer-based systems designed to support groups of people working together providing a shared environment. Given that developing this kind of application is not a trivial task because of the huge amount of time wasted on implementing infrastructure aspects, a few component-based approaches appeared. Groupware Workbench structures groupware using components and tools that encapsulate the technical difficulties of distributed and multi-user systems based on the 3C Collaboration Model. In this paper we propose the development of a Collablet product line using the Groupware Workbench. This approach combines the benefits of Software Product Lines and software components providing a systematic way for tailoring customized groupware through the use of Collablets automatically derived from product lines.

Keywords: software product lines, component-based development, groupware.

1 Introduction

Groupware development isn't a trivial task because it shares a particular set of common requirements [1] and on the top of that it needs specific software development techniques. One key issue that should be mitigated in groupware development is the huge amount of time wasted on implementing infrastructure aspects like protocols, synchronism, session management and others, leaving little time for implementing innovative solutions [2]. Following this line of thought, [3] cite several groupware component-based approaches allowing software reuse and making groupware development faster.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 446–450, 2010.

The Groupware Workbench [4] structures collaborative systems using components (Collablet Elements) and tools (Collablets) that encapsulate the technical difficulties of distributed and multi-user systems based on the 3C Collaboration Model [5, 6]. This model considers that collaboration is achieved by the interplay of communication, coordination and cooperation efforts. The 3C Collaboration Model guides all the development process from the domain analysis, organizing the feature model, through implementation where Collablet Elements are developed and classified according to the model.

In this paper we propose the development of a Collablet product line using the Groupware Workbench combining the benefits of SPL and software components. Our goal is to provide a systematic way for tailoring customized groupware through the use of Collablets derived from product lines. Collablets result from the joining of Collablet Element followed by adapting them to provide specific functionality.

2 A Discussion Forum Collablet Product Line

Discussion forum is an asynchronous textual communication tool, largely used to delve deeper into a subject of study. It is used in many different contexts and purposes, from entertainment where users discuss some topic of interest like TV shows, music and more, to education where students can "share their thinking with each other, comment on each other's ideas and find partners that share interests in order to get into a deeper discussion" [7].

Different uses for discussion forums impose different requirements. This section describes the development of a product line for developing discussion forums as Collablets – from the domain analysis until the instantiation of a product and installation on a groupware environment developed using Groupware Workbench.

2.1 Domain Analysis

In order to capture commonalities and variabilities on the discussion forums domain, it is necessary to know in advance their most common uses. Bellow, we describe three different scenarios of use for discussion forums:

- **Scenario 1 – General Purpose Forum.** General purpose forums are widely available on the internet and are open to user participation. In these forums, the discussion topic can vary from games to technical computer issues and the purpose of the discussion can be for entertainment, education, work or other. In general, there are no mediators, and all users play the same role, posting and answering posts to each topic. Some of these forums require user registration for keeping user data and future announcements.
- **Scenario 2 – Frequently Asked Questions (FAQ).** This kind of resource can be viewed as a discussion forum given that that users post their doubts and experts answer them. In this scenario, we can identify at least three different roles for the users: general users, mediators and expert users. General users are only allowed to post questions, but not to answer them. Mediators select posts to be answered by experts according to their relevance and check whether a

similar question has been answered. Expert users are the ones who answer questions and make them visible to other users.

- **Scenario 3 – Educational Forums.** In this kind of forum, topics are usually suggested by teachers or mediators, and topics are opened during a certain period of time for posts. Posts may be evaluated by teachers, depending on the educational methodology that is being applied. Like in scenario 2, here we can also identify different roles for the participants (students, mediators, teachers, and others). In this situation, a new set of requirements like session manager and categorization of messages should be taken into account.

Although having different objectives, applications for the scenarios abovementioned share a common set of characteristics. They are applications for posting and answering messages. These messages are displayed according to some criteria (hierarchically or by order of post) depending on each situation.

This way, based on the requirements of these scenarios and on our experience on developing and using groupware, we indentified a set of common and variable features for discussion forums development. These features were analyzed and classified according to the 3C Collaboration Model. One issue that should be observed is that although discussion forums are communication-oriented, most of the identified features are coordination- and cooperation-oriented, reflecting the intra-relationships of these dimensions of the collaboration model.

The identified features were then organized in the Forum PL 3C feature model, which is depicted in Fig 1. The feature model not only shows the features with their respective variability information (mandatory or optional), but also their purpose. Once again, the purpose of the derived products (communication) is indicated by the root of the feature model.

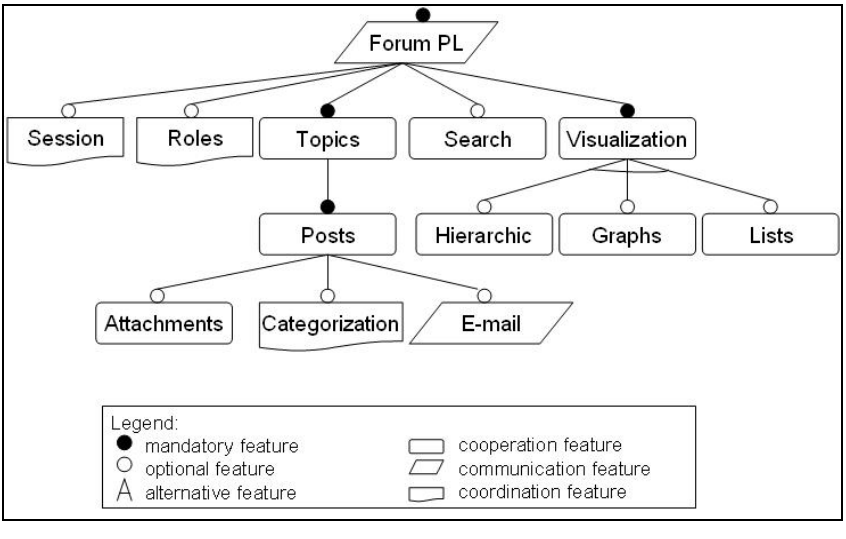


Fig. 1. Forum PL 3C-Feature Model

2.2 Design and Implementation

In order to provide a flexible architecture to support the variability provided by the Forum PL, the features identified in the previous phase were designed and implemented as independent Collablet Elements in Groupware Workbench. This assures that these components may be tailored and reused not only on Forum PL, but in any other product line or component-based software implemented with GW.

The independent Collablet Elements must be implemented and deployed on GW according to its 3C purpose. It keeps the workbench organized for future software maintenance and reuse, in addition to maintaining consistence with the previous development phase.

Product derivation [8] in the software product line engineering refers to the process of constructing a product from a set of reusable assets. In the Forum PL, products are derived by the composition of reusable Collablet Elements and configuration of specific Collablets. These artifacts encapsulate the technical difficulties of distributed and multi-user systems based on the 3C Collaboration Model.

2.3 Product Derivation

In order to facilitate the selection and composition of Collablet Elements and configuration of the Collablets, we specialize the GenArch [9] product derivation tool to incorporate a new model entitled Collab-specific Model.

The product derivation process starts by configuring the feature model. It encompasses the selection of Collablet features that satisfy the requirements of a certain product requested by customers. Based on the previous defined feature model configuration, the GenArch derivation process is devised in three steps: (i) selection of the Collablet Elements that will compose the specific Collablet; (ii) selection of the implementation elements (class, aspects, files, components, folders) that will be part of the derived product; and (iii) customization of Tool Descriptor Files – XML files that declare the Collablet Elements and their settings.

The selection of Collablet Elements from the Groupware Workbench is accomplished based on the configuration knowledge provided by the configuration model, which relates Collablet features to Collablet Elements. After that, the GenArch tool uses the information provided by the Collab-specific model that relates the Collablet Elements with implementation elements; in order to decide which implementation elements (classes, interfaces, extra files, etc.) will be part of the final product.

3 Conclusion

In this paper, we proposed the development of a Collablet Product Line (CPL) using the GW aiming to provide a systematic way for tailoring customized groupware derived from product lines. As an example, we developed a CPL for discussion forums.

The use of GW in the development of the product line was adequate, since the structure provided by GW was already designed for the reuse of software components. In addition, the GW already provides mechanisms for composition of Collablet Elements to the creation of Collablets and composition of Collablets for groupware. The concept of product line systematizes the groupware development process using

the GW, supplying the need of having technical management aspects that are important throughout the life cycle of the software.

This paper addresses ongoing research on the GPL development. We are currently investigating how to combine and instantiate two or more Collablets Product Lines in order to provide customizable Collablets on groupware composition according to specific group dynamics and needs.

Acknowledgments

This research is partially supported by project "Modelagem Computacional de Sistemas Biológicos e Sociais Baseada em Sistemas Multiagentes" from CNPq, num. 550865/2007-1. Bruno Gadelha, Elder Cirilo, Hugo Fuks and Carlos J. P. Lucena receive grants from CNPq. Hugo Fuks and Carlos J. P. Lucena also receive grants from FAPERJ.

References

1. Tietze, D.A.: A Framework For Developing Component-Based Co-Operative Applications. Ph.D. Dissertation, Technischen Universität Darmstadt, Germany (2001)
2. Greenberg, S.: Multimedia Tools and Applications, vol. 32(2), pp. 139–159 (February 2007) ISBN 1380-7501
3. Gadelha, B., Nunes, I., Fuks, H., Lucena, C.J.P.: An Approach for Developing Groupware Product Lines (GPL) based on the 3C Collaboration Model. In: CRIWG 2009, pp. 328–343 (2009)
4. Gerosa, M.A., Fuks, H.: A Component Based Workbench for Groupware Prototyping. In: 1st Workshop on Software Reuse Efforts (WSRE), 2nd Rise Summer School (2008)
5. Ellis, C.A., Gibbs, S.J., Rein, G.L.: Groupware - Some Issues and Experiences. Communications of the ACM 34(1), 38–58 (1991)
6. Fuks, H., Raposo, A., Gerosa, M.A., Pimentel, M., Lucena, C.J.P.: The 3C Collaboration Model. The Encyclopedia of E-Collaboration, Ned Kock (org), 637–644 (2007)
7. Gerosa, M.A., Filippo, D., Pimentel, M., Fuks, H., Lucena, C.J.P.: Is the Unfolding of the Group Discussion Off-Pattern? Improving Coordination Support in Educational Forums Using Mobile Devices. Computers and Education 54(2), 528–544 (2010)
8. Czarnecki, K., Eisenecker, U.W.: Generative programming: methods, tools, and applications. Addison-Wesley, USA (2000)
9. Cirilo, E., Kulesza, U., Lucena, C.: A Product Derivation Tool Based on Model-Driven Techniques and Annotations. JUCS 14, 1344–1367 (2008)

Towards Consistent Evolution of Feature Models

Jianmei Guo and Yinglin Wang

Department of Computer Science and Engineering,
Shanghai Jiao Tong University, Shanghai 200240, China
{guojianmei,ylwang}@sjtu.edu.cn

Abstract. This paper explores the possibility of consistent evolution of feature models (FMs), which should resolve the requested changes and maintain the consistency of FMs. According to the definition of FMs, we first analyze the primitive elements of FMs and suggest a set of atomic operations on FMs. Then we analyze and apply the semantics of change to FMs to support consistency maintenance during FMs evolution. The resolution of a requested change to an FM requires obtaining and executing a sequence of additional changes derived from the requested change for keeping the consistency of the FM. Our approach limits the consistency maintenance of an FM in a local range affected only by the requested change instead of the whole FM, which reduces the effort and improves the efficiency for the evolution and maintenance of FMs.

1 Introduction

Software product line (SPL) engineering captures commonalities and variabilities of an SPL in terms of *features* and documents them in a *feature model* (FM) [3,2,4]. SPLs and their FMs evolve continually with various unanticipated stakeholder requests and dynamic changes in the runtime environments of SPLs. Their evolution and consistency maintenance are still open issues.

Some researchers classified FMs evolution via changes as refactorings, specializations, and generalizations [7,10,11]. Such classification explains how changes to an FM have altered the products of an SPL [11]. It represents the types of FMs evolution at a high level but cannot reflect various concrete change operations during the evolution process. How one changes an FM X into a target FM Y using a sequence of sound operations is not obvious.

Many approaches were proposed to support automated analysis of FMs. They mostly use SAT solvers [5,11], BDD tools [8], or CSP solvers [9] to automate various reasoning tasks, e.g., checking satisfiability, detecting "dead" features, computing commonalities. They, however, suffer from the NP-hard problem of feature combinatorics and thus take a long time to perform with large FMs [6].

This paper explores the possibility of consistent evolution of FMs. We apply and extend techniques from ontology evolution [12] to FMs evolution. The consistent evolution of an FM is defined as the timely adaptation of the FM to the requested changes and the consistency maintenance for those changes. The consistency of FMs is regarded as an agreement among the features in terms of the semantics of FMs.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 451–455, 2010.

Our main contributions are as follows. First, we analyze the primitive elements of FMs and propose a set of atomic operations on FMs (see Section 2). Second, our approach analyzes and applies the semantics of change and clarifies FMs evolution by a sequence of changes (see Section 3). Third, our approach limits the consistency maintenance of an FM in a local range affected only by the requested change instead of the whole FM, which reduces the maintenance cost and improves the management efficiency.

2 Defining Atomic Operations on FMs

An FM is organized hierarchically and is graphically depicted as a *feature diagram* [1]. By integrating many former definitions of FMs [1,5,8,11], we adopt the notation as Fig. 1 shows (The example is inspired from [2,9]). An FM is a tree of features [8]. Every node in the tree has one parent except the *root feature* ('r: HIS'). A *terminal* or *concrete* feature (e.g., 'f4') is a leaf and a *non-terminal* or *compound* or *abstract* feature (e.g., 'f1') is an interior node of a feature diagram [5,11]. Connections between a feature and its group of children are classified as *And-* (e.g., 'f1', 'f2', and 'f3'), *Or-* (e.g., 'f10' and 'f11'), and *Alternative*-groups (e.g., 'f12', 'f13', and 'f14'). The members of And-groups can be either *mandatory* (e.g. 'f1') or *optional* (e.g. 'f3'). Or-groups and Alternative-groups have their own *cardinalities* [8]. Additional constraints comprise *requires* and *excludes* relationships [1], e.g., 'f4 requires f7'.

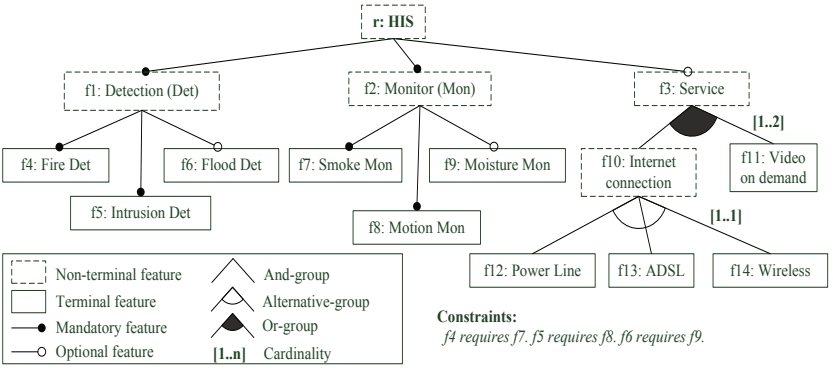


Fig. 1. Partial FM for the Home Integration Systems SPL

Based on the above definition of FMs, we can analyze the primitive elements of FMs. Since each element of FMs can be changed by one of the meta-change transformations [13], we suggest a set of atomic operations on FMs in Table 1. These operations are defined by the cross product of the set of FM elements and the set of meta-changes ('Add', 'Remove', and 'Set'). They represent the changes to FMs at the lowest level of complexity and can compose various complex changes.

Table 1. Atomic operations on FMs

Primitive Elements/Meta-Changes			Entity Operation		Attribute Operation
			Add	Remove	Set
Entity	Node	Non-terminal Feature	AddNF(new-nf, old-node)	RevNF(old-nf)	
		Terminal Feature	AddTF(new-tf, old-nf)	RevTF(old-tf)	
	Group	Feature Group	AddFG(new-fg, old-nf)	RevFG(old-fg)	
	Link	Parent Link	AddPL(new-pl, start-node, end-node)	RevPL(old-pl)	
		Requires Link	AddRL(new-rl, start-node, end-node)	RevRL(old-rl)	
		Excludes Link	AddEL(new-el, start-node, end-node)	RevEL(old-el)	
Attribute	Name (for all entities)				SetName(entity, 'name')
	Group Type (for all groups)				SetGT(fg, 'And/Or/Alternative')
	Optionality (for AND group members)				SetOpt(node, 'Mandatory/Optional')
	Cardinality (for OR/Alternative groups *)				SetCard(fg, 'mincard', 'maxcard')

* Every Alternative group has the fixed cardinality [1..1].

3 Semantics of Change to FMs

The evolution of FMs can be seen as a sequence of changes to FMs. Such changes can be composed of atomic operations on FMs. A change to FMs can be seen as a surjective mapping between FMs. As is shown in Fig. 2, given an FM and a requested change Ch, the application of the change Ch to the FM results in another FM', i.e., $FM' = Ch(FM)$, under $preconditions(FM, Ch) = true \wedge postconditions(FM', Ch) = true$. Here, *preconditions* of a change comprise a set of assertions that must be true to be able to apply the change, while *postconditions* of a change comprise a set of assertions that must be true after applying a change.

Since the application of a single change will not always leave an FM in a consistent state, it often derives a series of additional changes. Hence, the resolution of the requested change requires obtaining and executing these derived changes to maintain the consistency of the FM. If we decide whether a given FM is consistent or not by a decision function *consistency(FM)*, then:

Definition 1. Given an FM and a requested change Ch, the semantics of change to FM is defined as:

$$SemanticsOfChange(FM, Ch) = (Ch^1, ..., Ch^i, Ch^{i+1}, ..., Ch^{n-1})$$

where:

- FM is a given consistent FM, i.e., $consistency(FM) = true$;
- Ch is a requested change that can be applied to the FM, i.e., $preconditions(FM, Ch) = true$;
- $FM^1 = Ch(FM)$ is an FM representing the result of applying the requested change Ch to the FM, i.e., $postconditions(FM^1, Ch) = true$;
- $Ch^i, 1 \leq i \leq n-1$, is a derived change that satisfies the following set of conditions:

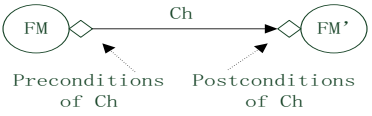


Fig. 2. Applying a change *Ch* to an FM

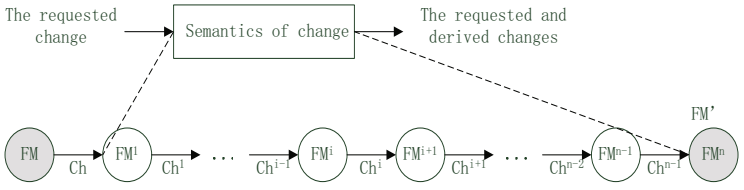


Fig. 3. The semantics of change to an FM

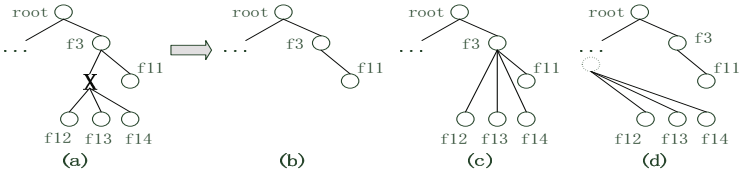


Fig. 4. Different evolution strategies. (a) Applying the $RevNF(\text{“}f10\text{”})$ alone to the FM shown in Fig. 1. (b) All child features are removed. (c) All child features are reconnected to the parent. (d) All child features are reconnected to some feature.

- $FM^{i+1} = Ch^i(FM^i)$, which implies that $preconditions(FM^i, Ch^i) = true$ and $postconditions(FM^{i+1}, Ch^i) = true$;
- $consistency(FM^i) = false, 1 \leq i \leq n-1$, and $consistency(FM^n) = true$.

Thus, as is shown in Fig. 3, the result of applying the requested change *Ch* to the *FM* is the *FM'*: $FM' = FM^n = Ch^{n-1}(...Ch^{i+1}(Ch^i(...Ch^1(Ch(FM)))))$.

Next, how to find and organize these derived changes that resolve the requested change and maintain the consistency of the FM? It is impractical to demand engineers to track down and keep in mind all the changes that are pending. Hence, we adapt the procedural approach [12] to realizing the task automatically. Our approach comprises five steps: first, a requested change is represented as a series of atomic operations defined in Table 1; second, the illegal changes are prohibited by checking the preconditions of each change; third, according to the cause and effect relationships between changes, additional changes are derived from the requested change for keeping consistency; fourth, the execution order of the requested and derived changes is determined in terms of the evolution strategy; fifth, all the confirmed changes are applied to the FM. Here, an *evolution strategy* unambiguously defines the way in which a change will be resolved. Fig. 4 demonstrates three evolution strategies for resolving a change.

4 Conclusion

This paper explores the possibility of consistency evolution of FMs from a perspective of atomic operations and their semantics. We contribute a set of atomic operations and the semantics of change to the consistency maintenance of an evolving FM. Our approach limits the consistency maintenance of an FM in a local range affected only by the requested change and thus reduces the effort and improves the efficiency. It is well suited to the incremental management of FMs evolution. Next, we plan to apply our approach to a large-size SPL and obtain more comprehensive evaluation.

Acknowledgments. Funding was provided by NSFC (No. 60773088), 863 Program (No. 2009AA04Z106), and Shanghai Municipal S&T Commission (No. 08JC1411700).

References

1. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, CMU (1990)
2. Kang, K.C., Lee, J., Donohoe, P.: Feature-oriented product line engineering. IEEE Software 19, 58–65 (2002)
3. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading (2001)
4. Pohl, K., Bockle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
5. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
6. Batory, D., Benavides, D., Ruiz-Cortes, A.: Automated analysis of feature models: challenges ahead. Communications of the ACM 49, 45–47 (2006)
7. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice 10, 7–29 (2005)
8. Czarnecki, K., Wasowski, A.: Feature Diagrams and Logics: There and Back Again. In: SPLC 2007, Kyoto, Japan, pp. 23–34 (2007)
9. Benavides, D., Martin-Arroyo, P.T., Cortes, A.R.: Automated reasoning on feature models. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg (2005)
10. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring product lines. In: GPCE 2006, Portland, Oregon, USA, pp. 201–210 (2006)
11. Thum, T., Batory, D.S., Kastner, C.: Reasoning about edits to feature models. In: ICSE 2009, Vancouver, Canada, pp. 254–264 (2009)
12. Stojanovic, L.: Methods and Tools for Ontology Evolution. PhD. Dissertation. University of Karlsruhe (2004)
13. Huersch, W.: Maintaining consistency and behaviour of object-oriented systems during evolution. ACM SIGPLAN Notices 32, 1–21 (1997)

SOPLE-DE: An Approach to Design Service-Oriented Product Line Architectures

Flávio M. Medeiros[1], Eduardo S. de Almeida[2], and Silvio R.L. Meira[1]

[1] Federal University of Pernambuco (UFPE)
[2] Federal University of Bahia (UFBA)
{fmm2,srlm}@cin.ufpe.br, esa@dcc.ufba.br

Abstract. Software reuse is crucial for enterprises interested in software quality and productivity gains. In this context, Software Product Line (SPL) and Service-Oriented Architecture (SOA) are two reuse strategies that share common goals and can be used together to increase reuse and produce service-oriented systems faster, cheaper and customizable to specific customers. In this sense, this work investigates the problem of designing software product lines using service-oriented architectures, and presents a systematic approach to design software product lines based on services. The proposed approach provides guidance to identify, design and document architectural components, services, service compositions and their associated flows. In addition, an initial experimental study performed with the intention of validating and refining the approach is also depicted demonstrating that the proposed solution can be viable.

Keywords: Service-Oriented Architecture (SOA), Software Product Line (SPL), Software Architecture and Software Development Processes.

1 Introduction

Software reuse is a key factor for enterprises interested in reducing development costs and increasing software quality [1]. In this context, SPL and SOA are two reuse strategies that share common goals, i.e., they both support the reuse of existing software and capabilities during the development of new systems and encourage the development of flexible and cost-effective software systems [2].

In this way, SPL and SOA concepts can be used together with the purpose of increasing and systematizing reuse during the Service-Oriented Development (SOD) and producing service-oriented systems faster, cheaper and customizable to specific customers [3]. Moreover, some service characteristics, e.g., dynamic discoverability and binding, can be used to support the development of Dynamic Software Product Lines (DSPL) [4].

This work investigates the problem of designing Service-Oriented Product Line Architectures (SO-PLA). This combination raises several challenges, such as how to identify and design services for the domain, decide the variation points to be considered in the context of SOD, identify service variability implementation mechanisms and define architectural views to represent the SO-PLA.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 456–460, 2010.

In this sense, a systematic design approach with a set of activities, with clearly defined inputs and outputs, and performed by a predefined set of roles is described in this work with the purpose of providing guidance to solve the problems of designing a SO-PLA. A service-oriented product line is considered as a set of similar service-oriented systems that supports the business processes of a specific domain and can be developed from a common set of core assets [5].

In order to define a SO-PLA, an approach is essential to provide guidance to the team, specify the artifacts to be produced, and associate activities with specific roles and the team as a whole. Without it, the development team may develop software in an ad-hoc manner, with success relying on the efforts of a few dedicated individual participants [6].

2 Related Work

An approach for developing service-oriented product lines was presented in [4]. In this work, a method to identify services and service compositions from feature models is depicted. In our work, we also provide methods to identify service candidates, not only from feature models, but also using business processes, use cases and quality attributes scenarios. In addition, some architectural views are proposed to represent the interactions among architectural elements.

The concept of Business Process Line (BPL) is used in [3]. This work provides a process to develop service-oriented product lines based on business processes that contain variability and can be customized to specific customers. Our work also considers variability in the business processes, but we also use feature models to represent variability in an easy and exploitable way.

In [7], an initial process for service-oriented product lines is presented. This work discusses the characteristics of SOA and SPL processes, but a systematic process for service-oriented product lines is not provided. The key difference of our work is the systematization of design. In addition, our approach was validated and refined through an initial experimental study.

3 The Proposed Approach (SOPLE-DE)

The SOPLE-DE is a top-down approach for the systematic identification, design and documentation of service-oriented core assets supporting the non-systematic reuse of SOD. It is divided in two cycles as SPL engineering. The core asset development cycle aims to provide guidelines and steps to identify, design and document architectural elements with variability. In the product development cycle, the architectural elements are specialized to a particular context according to specific customer requirements [5].

The SOPLE-DE considers the architectural style shown in Figure 1. This architectural style presents the layers that are commonly used in SOA [8]. Thus, SOPLE-DE provides guidelines to identify, design and document architectural elements for these layers. We use this architectural style because we believe that these layers are essential for any SOA solution.

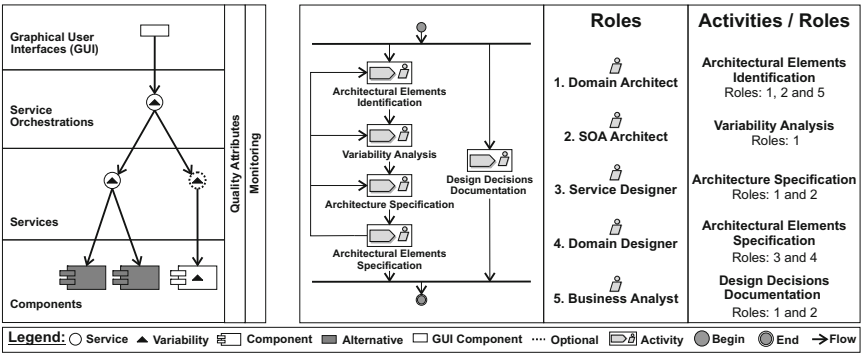


Fig. 1. Layered Architectural Style / SOPLE-DE Activities and Roles

The SOPLE-DE considers that a SO-PLA supports two variability levels as described next [3]: *Configuration variability,* in which architectural elements are selected from the core assets in order to obtain the target system, i.e., optional and alternative architectural elements are selected or excluded; *Customization variability,* in which architectural elements already selected for the architecture are customized according to specific requirements, i.e., architectural elements with variability are customized internally. SOPLE-DE also considers variability in the communication among the architectural elements, e.g., different protocols can be used for communication and the messages exchanged can be sent in a synchronous or asynchronous way.

The activities and roles of the SOPLE-DE are presented in Figure 1. It starts with the architectural elements identification activity, which receives the domain feature model, the business process models and the quality attribute scenarios as mandatory inputs. The domain use cases are optional inputs. It produces a list of components, services and service orchestration candidates for the SO-PLA. Moreover, the communication flows among these elements are also defined.

Subsequently, there is the variability analysis activity. It receives the list of components, services, service orchestrations and their flows identified previously, and defines and documents key architectural decisions regarding variability. In this activity, it is defined how the variability will be implemented. It refines the architectural elements identified previously.

Architecture specification is the next activity, in which the architecture is documented using different views in order to represent the concerns of the different stakeholders involved in the project [9]. An architecture is a complex entity that should be represented and documented upon several views (see Figure 2).

In the architectural elements specification activity, the low-level design of components and services is performed. SOPLE-DE suggests some UML diagrams to document the internal behavior of the architectural elements [10]. In parallel with these four activities described, the design decisions documentation activity is performed concurrently. In this activity, important design decisions, such as the selection of technologies and variability mechanisms, are documented.

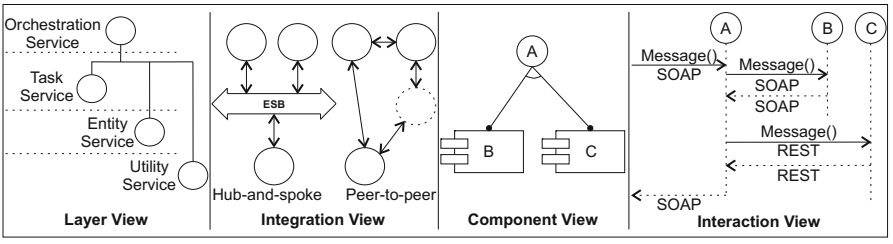


Fig. 2. Architectural Views

4 Experimental Study

An experimental study on the *Travel Reservation* domain was performed with the purpose of evaluating and refining the SOPLE-DE. In this experiment, the process of Wohlin [11] was used to define, plan and execute the experimental study. In addition, the Goal Question Metric (GQM) framework was also used to define the experiment [12]. The goal of this experiment was to *analyze the SOPLE-DE* for the purpose of *evaluation* with respect to its *efficacy* from the point of view of *researcher* in the context of *service-oriented product line projects*.

After collecting the information about the service coupling, instability and cohesion, the data collected was analyzed. Figure 3 shows the metric results for the services identified by the subjects. In the graphics, the axis (X) shows the ID of the subjects, while the axis (Y) represents the service coupling mean, service instability mean and the average cohesion of the service operations.

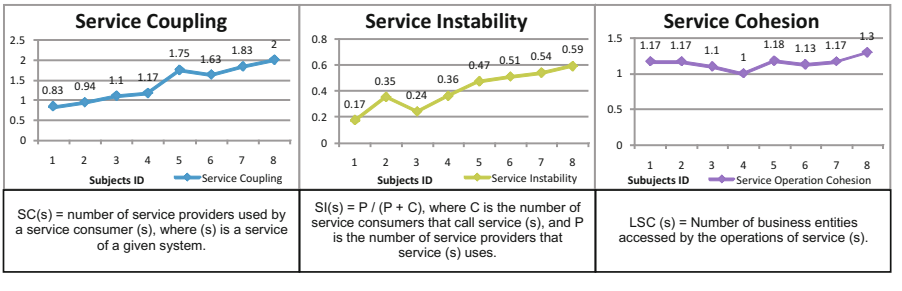


Fig. 3. Metric Results

The subjects with Id = 1, 2, 3 and 4 used the SOPLE-DE during the experiment, while subjects with Id = 5, 6, 7 and 8 designed the project without following a structured method. As it can be seen in Figure 3, the coupling, instability and cohesion of the services generated using the SOPLE-DE are lower when compared with the services identified by the subjects without use the method. The full description of this experiment can be found in [10].

5 Conclusions and Future Work

This work proposed an approach to design service-oriented product lines focusing on increasing reuse and flexibility, and supporting the development of service-oriented systems faster, cheaper and customizable to specific customers.

The SOPLE-DE approach was based on an extensive review of the available service-oriented processes, their weak and strong points and gaps in the area [10]. It can be seen as a systematic way to design service-oriented product line architectures through a well-defined sequence of activities with clearly defined inputs and outputs. Additionally, the approach was evaluated in an experimental study that presented findings that the SOPLE-DE can be viable to aid software architects to design service-oriented product line architectures with good coupling and instability, and identify services with cohesive operations.

Even it being a relevant contribution for the field, new routes need to be investigated in order to define a more complete process that consider all the software development disciplines, such as requirements, design and implementation, for product lines based on services. In addition, new experiments in different domains are necessary to gather more evidences about the efficacy of the proposed approach. Experiments in industry are also considered as future work.

References

1. Krueger, C.W.: Software reuse. ACM Computing Surveys 24(2) (1992)
2. Medeiros, F.M., de Almeida, E.S., Meira, S.R.L.: Towards an approach for service-oriented product line architectures. In: SOAPL 2009 (2009)
3. Boffoli, N., Caivano, D., Castelluccia, D., Maggi, F.M., Visaggio, G.: Business process lines to develop service-oriented architectures through the software product lines paradigm. In: SOAPL, pp. 143–147 (2008)
4. Lee, J., Muthig, D., Naab, M.: An approach for developing service-oriented product lines. In: SPLC, pp. 275–284. IEEE Computer Society, Los Alamitos (2008)
5. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading (2001)
6. Booch, G.: Managing the Object-Oriented Project. Addison-Wesley, Reading (1995)
7. Günther, S., Berger, T.: Service-oriented product lines: Towards a development process and feature management model for web services. In: SOAPL (2008)
8. Arsanjani, A.: Service-oriented modeling and architecture. Technical report, Service-Oriented Architecture and Web services Center of Excellence, IBM (2004)
9. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practices. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
10. Medeiros, F.M.: An approach to design service-oriented product line architectures. Master's thesis, Federal University of Pernambuco (2010)
11. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslen, A.: Experimentation in Software Engineering: An Introduction. Springer, Heidelberg (2000)
12. Basili, V., Caldiera, G., Rombach, D.H.: The goal question metric approach. In: Encyclopedia of Software Engineering. Wiley, Chichester (1994)

Multidimensional Classification Approach for Defining Product Line Engineering Transition Strategies

Bedir Tekinerdogan[1], Eray Tüzün[2], and Ediz Şaykol[2]

[1] Bilkent University, Department of Computer Engineering,
06800 Bilkent Ankara, Turkey
bedir@cs.bilkent.edu.tr
[2] Havelsan A.Ş., Peace Eagle Program, Research and Development Team,
ODTU Teknokent, 06531, Ankara, Turkey
{etuzun,esaykol}@havelsan.com.tr

Abstract. It is generally acknowledged that the transitioning process to a product line engineering approach is not trivial and as such requires a planned transition process. Different classifications of transition strategies have been proposed in the literature. It appears that these classification schemes are usually based on a single dimension. However, the adoption of a transition strategy is dependent on various criteria and very often it is not easy to characterize the required transition strategy. An appropriate characterization of the transition strategy is important for carrying out the right transition activities and steps to provide an operational product line engineering approach. In this paper, we first provide a conceptual model for defining the concepts related to transition strategies and then propose a multi-dimensional classification approach that aims to provide a more complete view on transition strategies.

Keywords: software product line engineering, product line transition strategy, multi-dimensional classification.

1 Introduction

It is generally acknowledged that transitioning to a product line engineering approach needs to be performed carefully to avoid failures and mitigate risks that are inherent to product line engineering and the transitioning process [7][9]. Likewise, the product line engineering community has proposed different transition strategies that aim to support the transition process and as such help to define a proper product line engineering approach for the organization. The selection of a transition strategy is largely driven by the specific business goals of the organization. To achieve these goals the organization needs to select one or more transition strategies that help to settle and carry out a product line engineering approach. Despite the benefits of each of these approaches we can observe that the transition strategies, as described in the literature, are classified differently, and likewise include somehow different transition strategies. Very often, it appears that the classification of the transition strategies is based on a single dimension. In reality, it seems that it is usually not easy to select a transition strategy based on the characterization of the existing, often complex state of

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 461–465, 2010.

a non-product line engineering approach. Yet, an appropriate characterization of the transition strategy is important for the selection of the right transition strategy and the healthy execution of the transition process.

In this paper we first provide a conceptual model that defines the key concepts for the transition process. The conceptual model represents a general model that can be used to instantiate multiple transition processes. Based on the conceptual model and the existing transition strategies, we propose a multi-dimensional classification approach for defining the space of transition strategies. The dimensions in this classification approach represent the criteria for classification while the values on the dimensions represent the separate transition strategies. The novel classification approach is complementary to existing approaches, reuses existing classification dimensions and proposes new classification dimensions to characterize a transition strategy. We believe that the analysis and the survey can support both practitioners and researchers. Practitioners will be supported by providing the first guidance in understanding the transition process and defining a more accurate transition strategy. For researchers the study may provide better insight in the current approaches and, if necessary, define new classification dimensions or new transition strategies.

The remainder of the paper is organized as follows. In Section 2 we provide the conceptual model for a transition strategy. Section 3 proposes the multidimensional classification approach for product line transition strategies. Finally, Section 4 concludes the paper.

2 Conceptual Model

The methodology and roadmap for switching to product line engineering from a traditional way of software development is defined as adoption or transition, and a plan of actions during this process is called the transition strategy [9]. Based on the existing literature [1][2][3][4][5][8][9][10] we have defined a conceptual model related to transition strategies in software product line engineering. The model is depicted in Figure 1. The rectangles represent concepts; the relations define association and inheritance relations similar to UML. In principle this conceptual model aims to represent the different transition strategies which can be found in the literature. A particular transition approach should be considered as an instance of this conceptual model.

3 Multidimensional Classification

An extensive review of the literature shows that there is actually no clear consensus yet on the transition strategies and the proposed transition strategies are defined from a particular standpoint only. Although the literature represents different classification mechanisms, based on the model as depicted in Figure 2 we can define the design space as a combination of multiple classification mechanisms or dimensions. Each dimension will have its own values or transition strategies. To identify the dimensions (classifications), initially we looked at the literature (domain) of the transition strategies. Unfortunately, deriving orthogonal dimensions from the domain is not as

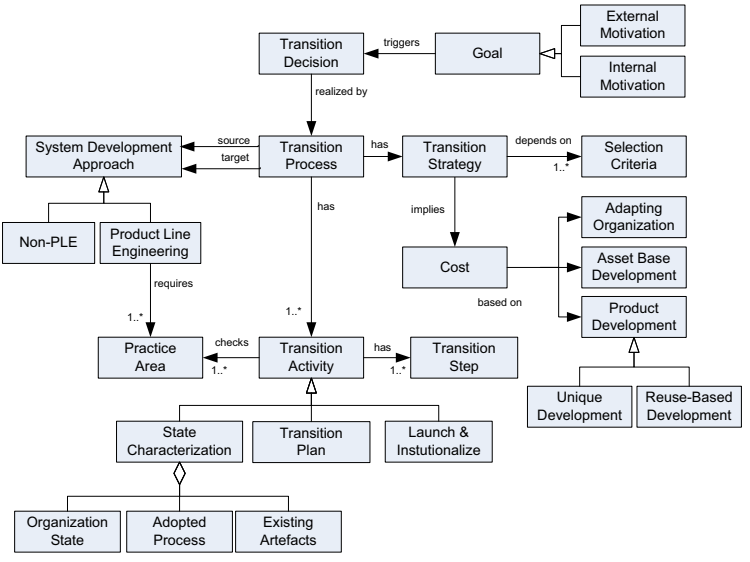


Fig. 1. Conceptual Model for Transition Strategies in Software Product Line Engineering

easy as it might look. The reason for this is that the existing classification approaches do not only use different dimensions and dimension names, but these also seem to overlap with each other. Very often the dimension is not explicitly defined but directly the transition strategies are listed.

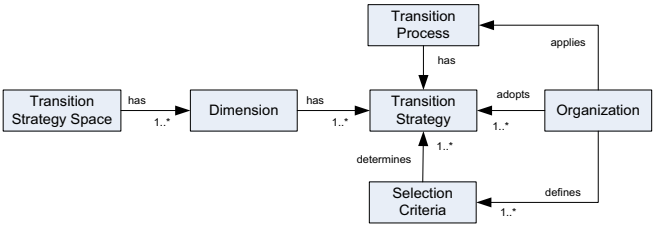


Fig. 2. Conceptual Model for Product Line Classification Schemes

Nevertheless, to illustrate the idea we have abstracted from the dimensions in the literature and defined and aligned a set of dimensions that we think are orthogonal and relevant for defining a space of transition strategies. These dimensions are defined in Table 1.

It should be noted here that Table 1 is a proposal for dimensions and as such we do not pretend that the dimensions are complete or fixed. New dimensions may be added to the list or the existing dimensions might be removed. Given the dimensions in Table 1, an organization can define its own transition strategy space that is spanned by a selected set of dimensions in the table. A coordinate in the space typically represents

Table 1. Description of multiple dimensions for classification of transition strategies

Dimension	Description	Transition Strategies
Required Effort	What is the required effort for the product line engineering transition process?	▪Lightweight ▪Heavyweight
Time of Adoption	When is the product line process adopted (e.g. from scratch, during development etc)?	▪Cold start ▪In Motion ▪Operational
Source of Products	What is the source for the development of product line artefacts?	▪Existing Products ▪Products developed from scratch
Target of Products	Which products are aimed to be included in asset base?	▪Refactor existing assets ▪New Assets
Approach	How is the product line engineering process adopted?	▪Pilot project ▪Incremental ▪Tactical ▪Big Bang
Anticipation	How is the product line scope defined?	▪Reactive ▪Proactive

the customized strategy that is defined from multiple perspectives (i.e. dimensions). As such, a transition strategy in this view is defined as a vector consisting of multiple values. For example, based on Table 1 we might derive the following transition strategy:

Transition Strategy Alternative = { (Required Effort.Lightweight, Time of Adoption.Cold Start, Source of Products.Products from Scratch, Target of Products. New Assets, Approach.Pilot Project, Anticipation.Proactive) }

Once the transition strategy space is defined, transition strategies can be selected. Based on a reflection and abstraction from the existing literature, in Table 2 we have compiled a set of selection criteria that we think are relevant for determining the transition strategy. An organization may derive a transition strategy by defining values for selection criteria. The nature and the exact number of selection criteria will be unique for each organization. In principle these selection criteria queries will reduce the transition space. To identify the feasible transition strategy for an organization sufficient selection criteria and their values must be defined.

Table 2. Description of selection criteria for adoption strategies

Selection Criteria	Description	Values
Maturity of the Organization	Is there a well-defined structure, are the communication channels well-defined, etc?	Low, Medium, High
Maturity of the application domain	How stable is the application domain?	Low, Medium, High
Maturity of the products	What is the quality of the products? How mature are the products?	Low, Medium, High
Maturity of practice areas	What is the degree of knowledge in the practice areas?	Low, Medium, High
Predictability of the product requirements	How easy is it to predict changes in the product requirements?	Not predictable, Changing, Fixed
Business goals	What are the high level costs that are of key interests to the business?	Time-to-market, Quality, Low Cost

4 Conclusions

In this paper, we have proposed a multi-dimensional classification approach that aims to provide a complementary and broader view on transition strategies. We have provided a conceptual model that summarizes the existing transition strategies. Based on the conceptual model and the study to existing transition strategies we have proposed a multi-dimensional classification approach for organizing the transition strategies. In our future work we aim to formalize the multidimensional approach and provide tool support for defining, selecting and prioritizing the dimensions and the selection criteria.

References

[1] Bayer, J., et al.: PuLSE: A Methodology to Develop Software Product Lines. In: Proc. Symposium Software Reusability (SSR 1999), pp. 122–131. ACM Press, New York (1999)
[2] Boeckle, G., Bermejo, J., Knauber, P., Krueger, C., Leite, J., van der Linden, F., Northrop, L., Stark, M., Weiss, D.: Adopting and institutionalizing a product line culture. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, p. 49. Springer, Heidelberg (2002)
[3] Bosch, J.: Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 257–271. Springer, Heidelberg (2002)
[4] Bühne, S., Chastek, G., Kakola, T., Knauber, P., Northrop, L., Thiel, S.: Exploring the context of product line adoption. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 19–31. Springer, Heidelberg (2004)
[5] Clements, P.C., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston (2002)
[6] Clements, P.C., Jones, L.G., McGregor, J.D., Northrop, L.M.: Getting there from here: A Roadmap for Software Product Line Adoption. CACM 49(12) (December 2006)
[7] IEEE Software, Special Issue of Software Product Lines (July/August 2002)
[8] Krueger, C.W.: New Methods in Software Product Line Development. In: Proc. of 10th Software Product Line Conference, BigLever Software, Austin, TX (2006)
[9] Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering – Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
[10] Schmid, K., Verlage, M.: The Economic Impact of Product Line Adoption and Evolution. IEEE Software 19(4), 50–57 (2002)

MARTE Mechanisms to Model Variability When Analyzing Embedded Software Product Lines

Lorea Belategi, Goiuria Sagardui, and Leire Etxeberria

Mondragon Unibertsitatea, Loramendi 4,
20500 Arrasate-Mondragón, Spain
{lbelategui,gsagardui,letxeberria}@eps.mondragon.edu

Abstract. Nowadays, embedded systems development is increasing its complexity dealing with quality among others. Model Driven Development (MDD) and Software Product Line (SPL) can be adequate paradigms to traditional development and validation methods. MARTE (UML Profile for Modeling and Analysis of Real-Time and Embedded systems) profile facilitates model analysis thus ensuring quality achievement from models. SPL requires taking into account variability like functional, quality attributes, platform and allocation. Therefore, variability mechanisms of MARTE profile have been studied in order to perform embedded SPL model analysis.

Keywords: MARTE, model analysis, software product lines, variability, embedded software.

1 Introduction

Embedded software architectures are usually complex and fragile, technological platforms evolve and change constantly and requirements such as reliability or safety add even more complexity to development.

Embedded systems distinguish themselves especially by following specific characteristics: heterogeneity (hardware/software), distribution (on potential multiple and heterogeneous hardware resources), ability to react (supervision, user interfaces modes), criticality, real-time and consumption constraints [1].

Embedded systems usually have to meet critical temporal requirements. Therefore, validation process must ensure not only functional requirements, but also non-functional ones, e.g., time requirements. Making embedded software validation is not trivial. Among other difficulties, in most cases, embedded software is hardware-dependent (the hardware imposes requirements on the software, which the software has to cope with) and may run under different configurations (communicating with different number and kind of devices).

MDD methodology abstracts from system complexity by the use of models where non-functional properties are attached in order to support validation through model analysis, simulations and testing. MARTE profile [1] standardized by OMG, allows performing schedulability and performance analysis based on models, suitable to validate single-product embedded systems where temporal aspects are critical.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 466–470, 2010.

On the other hand, embedded systems are favourable to be developed by SPL methodology due to their features; similar products with variability. Embedded SPL validation is much more complicated that in single-systems as variability in aspects related to validation must be taken into account: functional, quality attributes, platform, allocation and analysis variability [4]. Architecture assessment becomes crucial to ensure that the product line architecture is flexible enough to support different products and ensure quality attributes compliance.

This paper presents a study about the MARTE profile and its capability to annotate a SPL. The study has two phases: 1) identifying mechanisms the profile has to model variability and 2) analyzing which other existing mechanisms can be combined to fill the gaps in MARTE. Results of the study indicate the ability to perform embedded SPL model analysis with MARTE.

1.1 MARTE Elements for Analysis: *AnalysisContext*

MARTE analysis is intended to support accurate and trustworthy time related evaluations using formal quantitative analyses based on mathematical models [1]. Quantitative analysis techniques determine the output values such as response times, deadline failures, resource utilizations, etc. based on data provided as input; e.g., execution demands or deadlines.

Information from application model (where constraints, scenarios and software design is specified, including functional and quality requirements), platform model (where resources and their properties are described and platform design is specified) and software allocation (application to platform mapping) is required to perform model analysis. "*Extra annotations needed for analysis are to be attached to an actual design model, rather than requiring a special version of the design model to be created only for the analysis*" [1] by the use of stereotypes (that map model elements into the semantics of an analysis domain) and tagged values.

AnalysisContext identifies diagrams that gather information about systems behaviour and workload, execution platform and allocation for the analysis and specifies global parameters (properties that describe different cases being considered for analysis).

1.2 Variability in Model Analysis with MARTE

Variability is the key aspect of SPL that must be considered when analyzing models: not all products of the PL have the same functionalities; often, some of the hardware devices and other performance-affecting factors can vary from one product to another [10]; software can be allocated in different ways in a specific platform to optimize system objectives [3]; and two products with the same functionality may require different quality attributes, as well as the degree or priority of them [5]. Thus, analysis can vary from one product to another one. As a result, analysis process and *AnalysisContext* term must be extended to address SPL analysis.

2 Evaluation of Variability Mechanisms of MARTE

Although MARTE is not a profile for SPL it has some mechanisms such as *CombinedFragments*, abstract class, inheritance, interface implementation, variables,

Table 1. MARTE profile study related to variability in analysis stage

Variability Type	Justification	MARTE affected elements	MARTE Allows – Direct/Indirect way	Complementary proposals
Functional	Some functionality may vary from one product to another. Not all products have the same functionalities.	Application model (UML diagrams: collaboration, activity, sequence, interaction, component, class, composite structure, state machine and use case diagrams)	No. Although variables and *CombinedFragments* facilitate variability modelling in sequence diagram and UML abstract class, inheritance and interface implementation in class diagrams. It has no mechanism for variability management.	Variability profiles (PLUS, Ziadi) EAST-ADL2 (feature model and *ADLVariableElement*) CVL
Quality attributes (Performance and schedulability) - Optionality - Degree - Impact	Two products with the same functionality may require different quality attributes, as well as the priority or degree of them. Impacts may also arise among different quality attributes and/or among functionalities/ platforms/allocations and quality attributes.	Stereotypes of the MARTE profile	Although MARTE DataTypes (*ChoiceType* and *TupleType*) can help for quality attribute value, priority or degree is not specified and either optionality of the stereotypes.	Impacts by OCL [8] Optionality by feature model of EAST-ADL2
Platform	Variable platforms in design or in resources that make it up	Execution Platform model	Build different platforms (by submodels library) and configuring by parameters. UML interface implementation.	EAST-ADL2: *ADLHwElement* inherits from *ADLVariableElement*. SysML: Parametric diagrams. Variability profiles (PLUS, Ziadi) CVL
Allocation	Different software deployment in a specific platform	Allocation model	No. But it is possible to use variables for the link between application and service provided by the resource. The link between resource and service must be done manually.	CVL
Analysis	Not all products require the same analysis. Each product requires specific analysis that takes into account quality attribute, functional, platform and allocation variability.	*AnalysisContext* (This concept allows performing different analysis. It relates software behaviour and workload to execution platform)	*AnalysisContext* allows defining different analysis. In an *AnalysisContext* a scenario is analyzed under a platform. Case table to manage concrete values for the parameters of the *AnalysisContext*	EAST-ADL2 allows specifying analysis cases for each model element. Analysis composition view proposed in [2]

etc. that can help when analysing SPL models. There are profiles for variability annotation on UML models such as PLUS [6], Ziadi's UML profile for PL [13] and UML-F [9] UML profile for frameworks. CVL (Common Variability Language) is a variability specification language (still in development), that follows a separate language approach, and allows expressing variability in a base model and relationships between possible choices and the base model [7].

Other modelling languages that have been specified for embedded systems can also be complementary on SPL analysis.

SysML [11] complements UML with two new diagrams (requirements and parametric) and modifies some existing (activity, block definition and internal block).

Espinoza et al. [2] propose to use a similar diagram to SysML parametric diagrams in MARTE analysis for complex non-functional evaluation scenarios taking into account variations in the mapping from structure to architectural resources and parameterization i.e., propose to composite existing design models to experiment with different implementation or design decisions for the purpose of quantitative analysis.

EAST-ADL2 is a domain specific modelling language where functionalities are decomposed through different abstraction levels and development phases. It takes feature modelling as a reference and uses variation point concept.

Table 1 summarizes the results of the study of MARTE for variability support. Both MARTE mechanisms for variability and most relevant complementary modelling languages and approaches have been considered in the study.

3 Related Work

Tawhid and Petriu [12] propose a SPL modelling with functional variability and annotated with MARTE profile for performance. As stated in previous section Espinoza et al. [2] propose analysis composition view for analysis which can help on SPL analysis. There is a lack of how this approach may be linked to variability management or feature-oriented methods as it was a proposal for the analysis of a single-system.

4 Conclusions and Future Trends

This paper has analyzed MARTE profile and its capability to model variability. As MARTE was defined for single systems analysis, it can be combined with mechanisms from other profiles to tackle variability modelling and management for SPL.

Proposed modelling languages for embedded software can be complementary when SPL analysis. However, some modelling concepts can be overlapped creating an inconsistent modelling. A suitable combination of specific mechanisms of each modelling language is needed.

The next work to be carried out includes the definition of a management mechanism for all variability types identified for analysis. The combination of the studied mechanisms for embedded SPL analysis will be specified and new mechanisms will be proposed if necessary. A case study will be also performed to check the proposal and identify possible conflicts or problems.

Acknowledgments. This work was partially supported by OPTIMA (Basque Government under grants PI2009-1 and the Spanish Ministry of Science and Education under grants TIN2007-61779), by TESMO (OF157/2009) and VALINC (UE09+/99). It has been developed by the embedded systems group supported by the Department of Education, Universities and Research of the Basque Government.

References

1. UML Profile for Modeling and Analysis of Real-Time Embedded Systems. formal/2009-11-02 (2009)
2. Espinoza, H., Servat, D., Gérard, S.: Leveraging Analysis-Aided Design Decision Knowledge in UML-Based Development of Embedded Systems. In: SHARK 2008: Proc. of the 3rd Int. Workshop on Sharing and Reusing Architectural Knowledge, pp. 55–62. ACM, New York (2008)
3. Espinoza, H.: An Integrated Model-Driven Framework for Specifying and Analyzing Non-Functional Properties of Real-Time Systems, Thesis, DRT/LIST/DTSI/SOL/07-265/HE (2007)
4. Etxeberria, L., Sagardui, G.: Variability Driven Quality Evaluation in Software Product Lines. In: 12th International Software Product Line Conference (SPLC), pp. 243–252. IEEE, Los Alamitos (2008)
5. Etxeberria, L., Sagardui, G., Belategi, L.: Quality Aware Software Product Line Engineering. Journal of the Brazilian Computer Society (JBCS) 14 (2008)
6. Gomaa, H.: Designing software product lines with UML: From use cases to pattern-based software architectures. Addison Wesley Longman Publishing Co., Amsterdam (2004)
7. Haugen, Ø., Oldevik, B., Olsen, J.: Adding Standardized Variability to Domain Specific Languages. In: 12th International Software Product Line Conference, pp. 139–148. IEEE, Los Alamitos (2008)
8. UML 2.0 OCL Specification. ptc/03-10-14 (2003)
9. Pree, W., Fontoura, M., Rumpe, B.: Product Line Annotations with UML-F. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 188–197. Springer, Heidelberg (2002)
10. SEI: A Framework for Software Product Line Practice, Version 5.0 (2008)
11. OMG System Modeling Language (OMG SysML) V1.0. formal/2007-09-01 (2007)
12. Tawhid, R., Petriu, D.: Integrating Performance Analysis in the Model Driven Development of Software Product Lines. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 490–504. Springer, Heidelberg (2008)
13. Ziadi, T., Hélouët, L., Jézéquel, J.: Towards a UML Profile for Software Product Lines. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 129–139. Springer, Heidelberg (2004)

The UML «extend» Relationship as Support for Software Variability

Sofia Azevedo[1], Ricardo J. Machado[1], Alexandre Bragança[2], and Hugo Ribeiro[3]

[1] Universidade do Minho, Portugal
{sofia.azevedo,rmac}@dsi.uminho.pt
[2] Instituto Superior de Engenharia do Porto, Portugal
alex@dei.isep.ipp.pt
[3] Primavera Business Software Solutions, Portugal
hugo.ribeiro@primaverabss.com

Abstract. The development of software product lines with model-driven approaches involves dealing with diverse modeling artifacts such as use case diagrams, component diagrams, class diagrams, activity diagrams, sequence diagrams and others. In this paper we focus on use cases for product line development and we analyze them from the perspective of variability. In that context we explore the UML (Unified Modeling Language) «extend» relationship. This work allows understanding the activity of use case modeling with support for variability.

Keywords: use case, software product line, variability, «extend», alternative, option, specialization.

1 Introduction

Use case diagrams are one of the modeling artifacts that modelers have to deal with when developing product lines with model-driven approaches. This paper envisions use cases according to the perspective of variability. The *«extend»* relationship plays a vital role in variability modeling at the level of use cases and allows for the use case modeling activity to be applicable to the product line software development approach. This paper's contribute is on the understanding of the use case modeling activity with support for variability. We will illustrate our approach with some examples from the Fraunhofer IESE's GoPhone case study [1], which presents a series of use cases for a part of a mobile phone product line. We will propose different kinds of variability in use case diagrams.

The paper is structured as follows. Section 2 elaborates on the differences between others' approaches to variability modeling and ours. Section 3 analyzes the differences between our approach to modeling different types of variability with the UML (Unified Modeling Language) *«extend»* relationship and others' approaches. Section 4 illustrates our approach with some examples from the GoPhone. Finally Section 6 provides for some concluding remarks.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 471–475, 2010.

2 An Outline of Software Variability Modeling

Despite use cases being sometimes used as drafts during the process of developing software and not as modeling artifacts that actively contribute to the development of software, use cases shall have mechanisms to deal with variability in order for them to have the ability to actively contribute to the process of developing product lines. For instance modeling variability in use case diagrams is important to later model variability in activity diagrams [2]. In this paper we shortly talk about alternative, specialization and option use cases as the representation of the three variability types we propose to be translated into stereotypes to mark use cases.

We consider that product line modeling shall be top-down (rather than bottom-up), which means that the product line shall support as many products as possible within the given domain. In [3] Bayer, *et al*. refer that all variants do not have to be anticipated when modeling the product line. In [4] John and Muthig refer to required and anticipated variations as well as to a planned set of products for the product line, which indicates that their approach to product line modeling is bottom-up. A bottom-up approach would consider that all the products from the product line are known *a priori*. Bragança and Machado [5] represent variation points explicitly in use case diagrams through extension points. Their approach to product line modeling is bottom-up because they comment *«extend»* relationships with the name of the products from the product line on which the extension point shall be present.

In [4] John and Muthig refer the benefits of representing variability in use cases. Although we totally agree with the position of these authors towards those benefits, we cannot agree when they state that information on whether certain use cases are optional or alternatives to other use cases shall only be in decision models as it would overload use case diagrams and make them less readable. Our position is that features as well as use cases shall be suited for treating variability in its different types. Bachmann, *et al*. mention in [6] that variability shall be introduced at different phases of development of product families. If a use case is an alternative to another use case, then both use cases shall be modeled in the use case diagram, otherwise the use case diagram will only show a part of the possibilities of the possible products John and Muthig mention in [4].

Coplien, *et al*. defend in [7] the analysis of commonality and variability during the requirements analysis in order for the analysis decisions not to be taken during the implementation stage by the professionals who are not familiar with the implications and impact of decisions that shall be made much earlier during the development cycle. They refer that early decisions on commonality and variability contribute to large-scale reuse and the automated generation of family members.

Maßen and Lichter talk about three types of variability in [8]: optional, alternative and optional alternative (as opposite to alternatives that represent a “1 from n choice”, optional alternatives represent a “0 or 1 from n choice”). In this context they propose to extend the UML metamodel to incorporate two new relationships for connecting use cases. Our approach considers options and alternatives as well but we introduce these concepts into the UML metamodel through stereotypes (we consider that the *«extend»* relationship is adequate for modeling alternatives and a stereotype applicable to use cases for modeling options).

3 Different Perspectives on the *«extend»* Relationship

Gomaa and Shin [9] analyze variability in different modeling views of product lines. They mention that the *«extend»* relationship models a variation of requirements through alternatives. They also model options in use case diagrams by using the stereotype *«optional»* in use cases. We adopt these approaches to alternatives and options but we elaborate on another form of variability (specializations, which we consider to be a special kind of alternatives; Gomaa and Shin refer specialization as a means to express variability in [9]). Besides alternative and optional use cases, Gomaa and Shin consider kernel use cases (use cases common to all product line members). Gomaa models in [10] kernel and optional use cases both with the *«extend»* as well as with the *«include»* relationships (our approach is towards modeling kernel and optional use cases independently of their involvement in either *«extend»* or *«include»* relationships).

Halmans and Pohl propose in [11] use cases as the means to communicate variability relevant to the customer. Halmans and Pohl consider that generalizations between use cases are adequate to represent use cases' variants. This is not our position. We recommend to use the *«extend»* relationship instead of the generalization relationship. Halmans and Pohl consider that modeling mandatory and optional use cases with stereotypes in use cases is not adequate because the same use case can be mandatory for one use case and optional for another. Again this is not our position. We consider that a mandatory use case is not mandatory with regards to another use case, rather it is mandatory for all product line members. We also consider that an optional use case is optional with regards to one or more product line members.

Fowler suggests in his book "UML Distilled" [12] that we ignore the UML relationships between use cases besides the *«include»* and concentrate on the textual descriptions of use cases. We completely agree with Fowler on the textual descriptions but we cannot agree with the rest. The *«extend»* relationship is needed in order to formalize at an early stage (the use case modeling) where variation will occur when instantiating the product line. Bosch, *et al.* mention in [13] the need for describing variability within different modeling levels such as the requirements one.

4 Variability Types in the GoPhone Case Study

We consider that the *«extend»* relationship is adequate for modeling alternatives and specializations, and a stereotype applicable to use cases for modeling options. The three variability types we propose to be translated into stereotypes to mark use cases are: alternative, specialization and option. From now on we either use the *«extend»* relationship without stereotypes or with one of the two stereotypes applicable to this relationship (depending on whether we are modeling alternatives or specializations). We propose the stereotypes *«alternative»*, *«specialization»* and *«option»* to distinguish the three variability types. We also propose the stereotype *«variant»* to mark use cases at higher levels of abstraction before they are realized into alternatives or specializations. The stereotypes *«alternative»* and *«specialization»* shall be applicable to the *«extend»* relationship for modeling alternatives and specializations

respectively, and the stereotype *«option»* shall be applicable to use cases that represent options. Figure 1 shows some examples of alternative, specialization and option use cases from the GoPhone's message sending functionality.

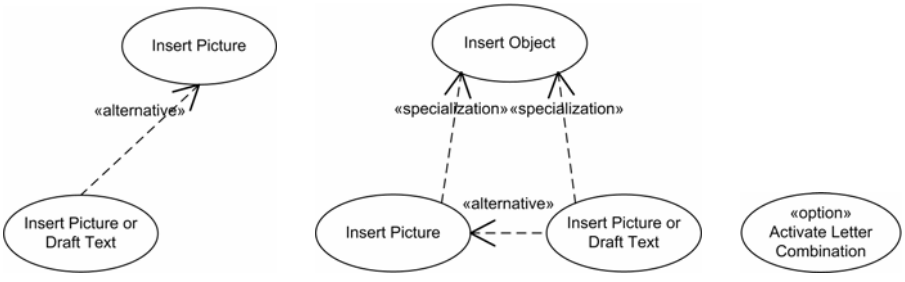


Fig. 1. Some examples of alternative, specialization and option variability types from the GoPhone's messaging domain

Consider that an extending use case is a use case that extends another use case and that an extended use case is a use case that is extended by other use cases. In the context of alternatives both extending and extended use cases represent supplementary functionality since both represent alternatives, which are not essential for a product without variability to function. If we have more than one alternative use case for the same functionality, one of those use cases shall be the alternative to all the others and extended by them. That use case is the one to be present in the products less robust in terms of functionality.

If the intention is to use differential specification, specializations shall be modeled with the *«extend»* relationship, otherwise they shall be modeled with the generalization relationship. Differential specification of specializations means that specialization use cases represent supplementary functionality regarding the use case they specialize, therefore a product without variability does not require the specialization use cases to function.

Options represent functionality that is only essential for a product with variability to function, therefore options represent supplementary functionality. However we do not recommend modeling options with the *«extend»* relationship because if the stereotype was on the relationship, the relationship itself would be optional and that is not the case (the use case is not optional with regards to any other use case, rather it is optional by itself). Options shall be modeled with a stereotype in use cases. The involvement of an option use case in either *«extend»* or *«include»* relationships, or even in none of those does not imply the presence of that use case in all product line members (which makes of it optional).

5 Conclusions

This paper has elaborated on the representation of variability in use case diagrams. It began by providing an analysis of the state-of-the-art concerned with this topic. Based on our position towards the related work we proposed three variability types to be

translated into stereotypes to mark use cases: alternative, specialization and option. We proposed both alternative and specialization use cases to be modeled with the *«extend»* relationship, and a stereotype applicable to use cases for modeling option use cases.

References

[1] Muthig, D., John, I., Anastasopoulos, M., Forster, T., Dörr, J., Schmid, K.: GoPhone - A Software Product Line in the Mobile Phone Domain, Fraunhofer IESE, IESE-Report No. 025.04/E (March 5, 2004)

[2] Bragança, A., Machado, R.J.: Extending UML 2.0 Metamodel for Complementary Usages of the «extend» Relationship within Use Case Variability Specification. In: 10th International Software Product Line Conference (SPLC 2006). IEEE Computer Society, Baltimore (2006)

[3] Bayer, J., Gerard, S., Haugen, Ø., Mansell, J., Møller-Pedersen, B., Oldevik, J., Tessier, P., Thibault, J.-P., Widen, T.: Consolidated Product Line Variability Modeling. In: Käköla, T., Duenas, J.C. (eds.) Software Product Lines - Research Issues in Engineering and Management, pp. 195–241. Springer, Heidelberg (2006)

[4] John, I., Muthig, D.: Product Line Modeling with Generic Use Cases. In: Workshop on Techniques for Exploiting Commonality Through Variability Management. Springer, San Diego (2002)

[5] Bragança, A., Machado, R.J.: Deriving Software Product Line's Architectural Requirements from Use Cases: An Experimental Approach. In: 2nd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES 2005). TUCS General Publications, Rennes (2005)

[6] Bachmann, F., Goedicke, M., Leite, J., Nord, R., Pohl, K., Ramesh, B., Vilbig, A.: A Meta-model for Representing Variability in Product Family Development. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 66–80. Springer, Heidelberg (2004)

[7] Coplien, J., Hoffman, D., Weiss, D.: Commonality and Variability in Software Engineering. IEEE Software 15, 37–45 (1998)

[8] Maßen, T.v.d., Lichter, H.: Modeling Variability by UML Use Case Diagrams. In: International Workshop on Requirements Engineering for Product Lines (REPL 2002), Avaya Labs, Essen (2002)

[9] Gomaa, H., Shin, M.E.: A Multiple-View Meta-modeling Approach for Variability Management in Software Product Lines. In: Bosch, J., Krueger, C. (eds.) ICOIN 2004 and ICSR 2004. LNCS, vol. 3107, pp. 274–285. Springer, Heidelberg (2004)

[10] Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley, Upper Saddle River (2004)

[11] Halmans, G., Pohl, K.: Communicating the Variability of a Software-Product Family to Customers. Software and Systems Modeling 2, 15–36 (2003)

[12] Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley, Upper Saddle River (2004)

[13] Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J.H., Pohl, K.: Variability Issues in Software Product Lines. In: van der Linden, F.J. (ed.) PFE 2002. LNCS, vol. 2290, p. 13. Springer, Heidelberg (2002)

Feature Diagrams as Package Dependencies*

Roberto Di Cosmo and Stefano Zacchiroli

Université Paris Diderot, PPS, UMR 7126, Paris, France
roberto@dicosmo.org, zack@pps.jussieu.fr

Abstract. FOSS (Free and Open Source Software) distributions use dependencies and package managers to maintain huge collections of packages and their installations; recent research have led to efficient and complete configuration tools and techniques, based on state of the art solvers, that are being adopted in industry. We show how to encode a significant subset of Free Feature Diagrams as interdependent packages, enabling to reuse package tools and research results into software product lines.

Keywords: software product lines, open source, package, component, feature diagram, dependencies.

1 Introduction

Feature models [11] are essential devices to reason about software product lines (SPLs). As features and their interdependencies get more complex, managing models quickly turns into a non-trivial task for humans. Researchers have worked to improve the situation by establishing connections between Feature Diagrams (FD) and notions like grammars [6], propositional logics [2], and constraint programming [3], paving the way to use automatic tools like SAT solvers [9,12] and proof assistants [10] for SPL configuration.

Meanwhile FOSS distributions like Debian, Red Hat, and Suse have used for the past 15 years packages and dependencies to maintain some among the largest software collections known: package managers are used daily to maintain millions of installations built by selecting components from repositories of tens of thousands packages. Recent research efforts [1,4,13,16] have led to the development of efficient and complete configuration and maintenance tools, as well as metrics for FOSS distributions, based on state of the art solvers, which are rapidly being adopted in industry.

We show how a significant subset of Free Feature Diagrams can be compactly encoded as interdependent packages, opening the way to massive reuse in SPLs of research results and tools coming from FOSS research. In particular, package management tools are able to scale up to tens of thousands components and hundreds of thousands dependencies, and cope well with component evolution, which is routine in FOSS. Both aspects may be of great interest for the SPL community.

* This work is partially supported by the European Community FP7, MANCOOSI project, grant agreement n. 214898.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 476–480, 2010.

2 Package Dependencies for Distribution Maintenance

In FOSS distributions a *package* is a bundle that ships a (software) component, the data needed to configure it, and metadata which describe its attributes and expectations on the deployment environment [7]. For simplicity, we focus on packages as used in the Debian distribution, but the discussion applies almost unchanged to other popular package formats like RPM (see [13] for details).

Here is a sample metadata excerpt from the `firefox` package:

```
Package: firefox
Version: 1.5.0.1-2 ...
Depends: fontconfig, psmisc, libatk1.0-0 (>= 1.9.0), libc6 (>= 2.3.5-1) ...
Suggests: xprint, firefox-gnome-support (= 1.5.0.1-2), latex-xft-fonts
Conflicts: mozilla-firefox (<< 1.5-1)
Replaces: mozilla-firefox
Provides: www-browser, ...
```

Every package has a version that is used to give a temporal order to the packaged component release. The kinds of relationships that can be expressed in the metadata of a package `p` are numerous, but the most important are:

- `Depends`: a list of package disjunctions $p_1 \mid \ldots \mid p_n, \ \ldots \ , q_1 \mid \ldots \mid q_m$, where each atom can carry a version predicate (e.g. $\geq 1.9.0$). For the owner package to be installable, at least one package in each disjunction must be installed.
- `Conflicts`: a list of package predicates $p_1, p_2, \ldots p_n$. For the owner package to be installable, none of the p_i must be installed. *Self conflicts are ignored.*
- `Recommends` similar to `Depends`, it indicates an optional dependency; it might be advisable to satisfy it, but it is not needed to obtain a working system.

A *repository* R is a set of packages; a subset $I \subseteq R$ of it is said to be a *healthy installation* if all dependencies of packages in I are satisfied, and none of the conflicts is. Precise formal meanings to all these notions have been given elsewhere (see [8,13] and the Mancoosi project `http://www.mancoosi.org`). Tools are available in the distribution world to choose healthy installations according to user requests [15] and to perform sophisticated repository analysis [1,5].

3 Encoding Feature Diagrams as Package Dependencies

We show how a core subset of Feature Diagrams (FD) can be compactly encoded as packages. Due to the differences among FD formalisms, we provide the encoding for a significant subset of Free Feature Diagrams (FFD) [14] that captures many known formalisms, and allows to claim that our encoding is of general interest.

FFD is a general framework that allows to capture different classes of FD by specifying a few parameters: the kind of graph GT (Tree or DAG); the node type NT (*and*, *or*, *xor*, or *opt*; the latter encoding explicit optionality within a node-based semantics [14]); the graphical constraint type GCT ($\Rightarrow$ for implication, and $\mid$ for mutual exclusion); and the textual constraint language TCL (usually including just implication and mutual exclusion, noted n *implies* n' and n *mutex* n', respectively).

Definition 1 (Free Feature Diagram). $d \in FFD(GT, NT, GCT, TCL) = (N, P, r, \lambda, DE, CE, \Phi)$ *where:*

- N *is a set of nodes*
- $P \subseteq N$ *is a set of primitive nodes*
- $r \in N$ *is the root node*
- $\lambda : N \to NT$ *labels each node with an operator from* NT
- $DE \subseteq N \times N$ *is the set of decomposition edges;* $(n, n') \in DE$ *is noted* $n \to n'$
- $CE \subseteq N \times GCT \times N$ *is the set of constraint edges*
- $\Phi \subseteq TCL$ *are the textual constraints*

A few well-formedness constraints are imposed: only r has no parent; the decomposition edges do not contain cycles; if GT is Tree, then DE forms a tree; nodes are labeled with operators of the appropriate arity.

Precise formal semantics of FFD is given in terms of valid models [14]:

Definition 2 (Valid model). *A* valid model *of a feature diagram* d *is* $M \subseteq N$ *such that: (a)* $r \in M$*, (b)* M *satisfies the operators attached to each node, as well as all the (c) graphical and (d) textual constraints, with the additional requirement that (e) if a node is in the model, then at least one of its parents (called the* justification*) is in the model too.*

We call FFD_{core} the fragment of FFD obtained by restricting the operators in NT to *or*, *and*, *xor*, *opt*, and the operators in GCT and TCL to *implies* and *mutex*; this fragment is enough to cover several well known Feature Diagram formalisms (OFT, OFD, RFD, VBFD, GPFT and PFT in the classification given in [14]).

We will now show how to encode any $d \in FFD_{core}$ as a package repository $R(d)$, so that valid models correspond to healthy installations of a specific package $p_r \in R(d)$.

Definition 3 (FFD_{core} encoding). *Let* $d \in FFD_{core}(GT, NT, GCT, TCL) = (N, P, r, \lambda, DE, CE, \Phi)$*, we define the package repository* $R(d)$ *as follows:*

- *the packages* P *are defined as* $\{(n, 1) | n \in N\}$*, so we have a package for each node in the diagram, with a unique version number,* 1
- $\forall$ *node* n *with sons* $n_1, \ldots, n_k$*, add dependencies as follows:*
  - *if* $\lambda(n) = or$*, add* $n_1, \ldots, n_k$ *as disjunctive dependencies for* n
  - *if* $\lambda(n) = and$*, add* $n_1, \ldots, n_k$ *as conjunctive dependencies for* n
  - *if* $\lambda(n) = xor$*, add* $n_1, \ldots, n_k$ *as disjunctive dependencies of* n*, and add* $n_1, \ldots n_{i-1}, n_{i+1}, \ldots, n_k$ *as conflicts for* $\forall n_i$
- $\forall$ *node* n *with son* n' *and* $\lambda(n) = opt$*, add* n' *as a recommend of* n
- $\forall$ *constraint* c *in* CE *or* Φ*, add the following dependencies:*
  - *if* c *is* $n \Rightarrow n'$ *or* n *implies* n'*, add* n' *as a conjunctive dependency of* n
  - *if* c *is* $n | n'$ *or* n *mutex* n'*, add* n' *to the conflicts of* n
- *if* $GT = Tree$*,* $\forall n \to n' \in DE$*, add* n *as a conjunctive dependency for* n'
- *if* $GT = DAG$*,* $\forall n \neq r$*, add a dependency on* $n_1 \mid \cdots \mid n_k$ *where* $n_1, \ldots, n_k$ *are all the parents of* n

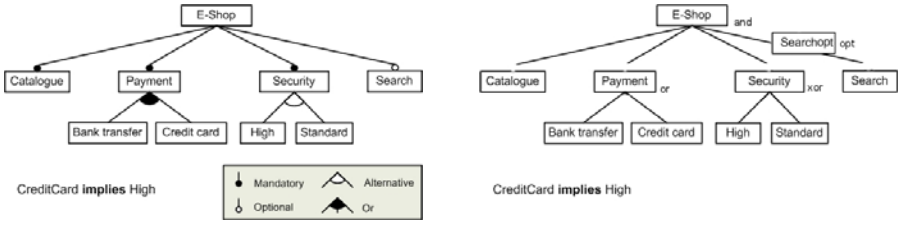


Fig. 1. Sample E-Shop feature model: as FD (on the left) and FFD (on the right)

An even more compact, linear-space encoding for justifications and *xor* nodes can be given using *virtual packages* [7], exploiting the property that self-conflicts are ignored; it has been omitted here due to space constraints.

Example 1. A feature model using an edge-based semantics for an e-shop is shown in Figure 1 as FD (on the left) and FFD (on the right). Its encoding as package repository is reported below, where we drop `Version: 1`.

```
Package: E-Shop
Depends: Catalogue, Payment, Security, SearchOpt

Package: Catalogue
Depends: E-Shop

Package: Payment
Depends: BankTransfer | CreditCard, E-Shop

Package: BankTransfer
Depends: Payment

Package: CreditCard
Depends: Payment, High

Package: Security
Depends: High | Standard, E-Shop

Package: High
Depends: Security
Conflicts: Standard

Package: Standard
Depends: Security
Conflicts: High

Package: SearchOpt
Depends: E-Shop
Recommends: Search

Package: Search
Depends: SearchOpt
```

Notice how all kinds of metadata are used: conflicts encode *mutual exclusion*, recommends encode optional features, conjunctive depends encode *and* nodes and implications, disjunctive dependencies encode *or* nodes. It is now possible to establish the key property of the detailed encoding.

Theorem 1 (Soundness and completeness). *A subset $M \subseteq N$ of the nodes of a $d \in FFD_{core}$ is a valid model of d if and only if m is a healthy installation for the package repository encoding $R(d)$.*

Proof. The proof is by case analysis on the definition of a valid model, and the structure of the encoding. Details are omitted due to lack of space.

4 Conclusions and Future Work

We have established a direct mapping from a significant subset of Free Feature Diagrams to packages of FOSS distributions. This paves the way to reuse of

theoretical results as well as tools coming from FOSS research. Package management tools scale to tens of thousands packages and hundreds of thousands dependencies, and cope with evolving components. For instance, the `edos.debian.net` site provides quality metrics for FOSS distributions comprising more than 20'000 packages and 400'000 dependencies, daily, since 2006. Also, model construction with respect to user-defined optimizations is implemented by several tools, and competitions like `www.mancoosi.org/misc-2010` are improving their efficiency.

We plan to extend the current encoding to all FFD constructs such as cardinality constraints and to validate the proposed approach by providing a full toolchain that attacks SPL problems using existing package management technology.

We hope that our work will contribute to establish a connection between SPLs and package management, for the joint benefit of both communities.

References

1. Abate, P., Boender, J., Di Cosmo, R., Zacchiroli, S.: Strong dependencies between software components. In: ESEM 2009, pp. 89–99. IEEE, Los Alamitos (2009)
2. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
3. Benavides, D., Martín-Arroyo, P.T., Cortés, A.R.: Automated reasoning on feature models. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg (2005)
4. Berre, D.L., Rapicault, P.: Dependency management for the Eclipse ecosystem. In: IWOCE 2009. ACM, New York (2009)
5. Boender, J., Di Cosmo, R., Vouillon, J., Durak, B., Mancinelli, F.: Improving the quality of GNU/Linux distributions. In: COMPSAC, pp. 1240–1246. IEEE, Los Alamitos (2008)
6. de Jonge, M., Visser, J.: Grammars as feature diagrams. In: ICSR7 Workshop on Generative Programming. pp. 23–24 (2002)
7. Di Cosmo, R., Trezentos, P., Zacchiroli, S.: Package upgrades in FOSS distributions: Details and challenges. In: HotSWup 2008. ACM, New York (2008)
8. EDOS project, WP2 team: Report on formal management of software dependencies. Deliverable Work Package 2, Deliverable 2 (2006)
9. Janota, M.: Do sat solvers make good configurators? In: SPLC 2008, Second Volume (Workshops), pp. 191–195 (2008)
10. Janota, M., Kiniry, J.: Reasoning about feature models in higher-order logic. In: SPLC, pp. 13–22. IEEE Computer Society, Los Alamitos (2007)
11. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. rep., CMU (1990)
12. Le Berre, D., Parrain, A.: On SAT technologies for dependency management and beyond. In: ASPL 2008 (2008)
13. Mancinelli, F., Boender, J., Di Cosmo, R., Vouillon, J., Durak, B., Leroy, X., Treinen, R.: Managing the complexity of large free and open source package-based software distributions. In: ASE 2006, pp. 199–208. IEEE, Los Alamitos (2006)
14. Schobbens, P.Y., Heymans, P., Trigaux, J.C.: Feature diagrams: A survey and a formal semantics. In: RE 2006, pp. 136–145. IEEE, Los Alamitos (2006)
15. Treinen, R., Zacchiroli, S.: Solving package dependencies: from EDOS to Mancoosi. In: DebConf 8: proceedings of the 9th conference of the Debian project (2008)
16. Tucker, C., Shuffelton, D., Jhala, R., Lerner, S.: OPIUM: Optimal package install/uninstall manager. In: ICSE 2007, pp. 178–188 (2007)

Visualizing and Analyzing Software Variability with Bar Diagrams and Occurrence Matrices

Slawomir Duszynski*

Fraunhofer Institute for Experimental Software Engineering (IESE), Fraunhofer-Platz 1,
67663 Kaiserslautern, Germany
Slawomir.Duszynski@iese.fraunhofer.de

Abstract. Software product lines can be developed in a proactive, reactive or extractive way. In the last case, an essential step is an analysis of the existing implementation of a set of similar software products to identify common and variable assets. If the variability across the similar products was not explicitly managed during their development, the information about it can be recovered with reverse engineering techniques. This paper proposes a simple and flexible technique for organizing and visualizing variability information, which can be particularly useful in the extractive product line adoption context. The technique can be applied to source code, models, and other types of product line artifacts. We discuss the advantages of using bar diagrams and occurrence matrices and demonstrate an example usage in an n-ary text diff.

Keywords: product lines, visualization, variability, reverse engineering.

1 Introduction

Adoption of the product line paradigm in a development organization can bring considerable advantages, such as reduced development cost and faster time-to-market. However, the need for disciplined reuse often becomes apparent only when software products already exist. For example, the software mitosis phenomenon [1] leads to creation of many similar copies of a software system, independently evolving from a common ancestor system. For embedded software, the mitosis can be driven by porting the software to many underlying hardware platforms. In response to the mitosis, the organization can follow the extractive product line adoption approach [2].

In software mitosis, the variability across the similar systems is introduced in a distributed, unmanaged way. Consequently, the exact variability distribution is usually unknown. However, detailed information on variability is indispensable, both to the creation of a product line from existing similar systems and to the subsequent product line evolution (e.g. product maintenance, adding new features, product derivation). To recover detailed variability information, using reverse engineering techniques on the source code of similar products has been proposed [4] [5].

* This work was performed within the Fraunhofer Innovation Cluster for Digital Commercial Vehicle Technology (DNT/CVT Cluster).

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 481–485, 2010.

In this paper we present a simple, yet flexible technique for organizing, visualizing and analyzing detailed variability information in a reverse engineering context. The technique supports efficient identification of core and variable parts in the analyzed assets, and enables versatile analyses of variability distribution. Unlike the high-level variability representations, such as decision models, feature models [3] or product maps [7], it provides sufficient details for common and variable asset extraction on the source code level. Section 2 describes the technique and its example usage in an n-ary diff, while Section 3 generalizes it. Section 4 gives outlook on the future work.

Related work. Venn diagrams [6] and Euler diagrams visualize relations between sets of elements. Feature models [3] specify product line variability on the semantic level. The occurrence matrix idea is inspired by product maps [7] and truth tables.

2 Occurrence Matrices and Bar Diagrams

To illustrate the idea of occurrence matrices and bar diagrams, we present an example: a diff technique capable of comparing n files at once. Consider three variants of a source code file: *A.c*, *B.c* and *C.c* (Table 1).

Table 1. Source code of the three example file variants

Source code of *A.c*	Source code of *B.c*	Source code of *C.c*

`#include "cal.h"`
`int f(int x)`
`{`
`  if(x > 1024)`
`   x = x % 1024;`
`  x += 3;`
`  x = cal(x, NO_A);`
`  return x;`
`}`
	`#include "cal2.h"`
	`int f(int x)`
	`{`
	`  x = cal2(x, NO_B);`
	`  if(x > 1024)`
	`   x = x % 1024;`
	`  return x;`
	`}`
		`#include "cal2.h"`
		`int f(int x)`
		`{`
		`  x = cal2(x, NO_C);`
		`  x += 3;`
		`  return x;`
		`}`

There exist some similarities across the three variants, as well as some differences. However, using the diff tool on each pair of the variants (Figure 1) does not deliver information directly useful for common and variable asset identification, such as the locations of common or variant-unique code parts. To identify common and variable code, further interpretation of the results by a human is required. This task needs some effort for three compared variants, and it would be much harder for a larger number of variants, since the number of compared pairs grows in a quadratic way.

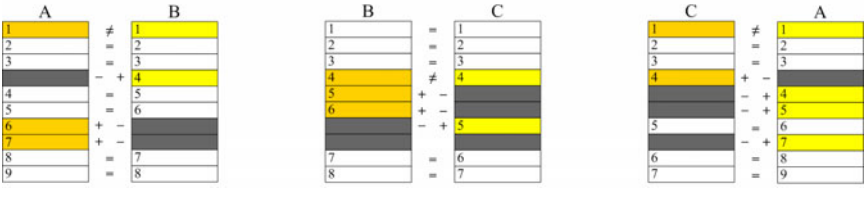


Fig. 1. Results of a standard diff tool for each pair of the three analyzed variants. The small rectangles represent code lines, and the numbers are line numbers in the respective file.

Occurrence matrices. To support the reasoning about common and unique code parts, we introduce a new technique of *occurrence matrices*. The matrices use the diff results and are constructed for each compared variant in the following way: the rows of the matrix represent the lines of the given file variant, while the columns represent all the compared variants. A field of the matrix has a value of *"1"* if the line represented by the field's row was found by diff as being equal to a line from the variant represented by the field's column, and a value of *"0"* otherwise. In addition, each matrix has a summary column which contains the number of occurrences of each line in all analyzed variants. Figure 2 shows the three occurrence matrices for files *A.c*, *B.c* and *C.c*, and a summary matrix. The rows of summary matrix represent the union of all lines of *A*, *B* and *C*. The union is constructed by first taking all the rows from the first variant, and then for each subsequent variant adding only these rows which were not found equal to any of the rows already in the matrix (for example, line 2 from file *B* is not added since the very same line is already represented by line 2 from file *A*). In the result, the summary matrix contains all the lines that exist in the analyzed code, but lines shared across the variants are counted only once.

A

	A	B	C	Sum
1	1	0	0	1
2	1	1	1	3
3	1	1	1	3
4	1	1	0	2
5	1	1	0	2
6	1	0	1	2
7	1	0	0	1
8	1	1	1	3
9	1	1	1	3

B

	A	B	C	Sum
1	0	1	1	2
2	1	1	1	3
3	1	1	1	3
4	0	1	0	1
5	1	1	0	2
6	1	1	0	2
7	1	1	1	3
8	1	1	1	3

C

	A	B	C	Sum
1	0	1	1	2
2	1	1	1	3
3	1	1	1	3
4	0	0	1	1
5	1	0	1	2
6	1	1	1	3
7	1	1	1	3

All

	A	B	C	Sum
A1	1	0	0	1
A2	1	1	1	3
A3	1	1	1	3
A4	1	1	0	2
A5	1	1	0	2
A6	1	0	1	2
A7	1	0	0	1
A8	1	1	1	3
A9	1	1	1	3
B1	0	1	1	2
B4	0	1	0	1
C4	0	0	1	1

Fig. 2. Occurrence matrices for A.c, B.c and C.c, and the summary matrix for union of all lines

The information stored in the occurrence matrices can be used in a number of ways:

- **Variability status information** for each line is given by the *"Sum"* column. The lines where the sum value is equal to the number of analyzed variants *n* are *core* lines that are identical in all variants. The lines with a value of *1* exist in only one variant and are *unique* to it. The lines with values in the *2..n-1* range are *shared* across some, but not all of the variants. Since the status of each line is known, the variability analysis on the source code level can be supported in a text editor by coloring or marking the code lines according to their status (Figure 3 left). Hence, the diff results are lifted to higher abstraction level of commonality and variability.

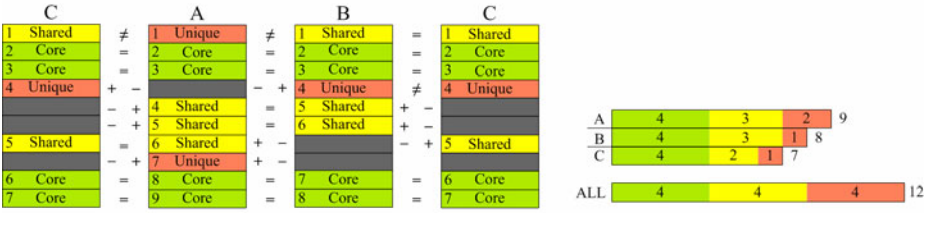


Fig. 3. Variability status shown for the lines of analyzed files A.c, B.c and C.c, with diff results in between for reference (*left*). Bar diagram constructed from the occurrence matrices (*right*).

- **Bar diagrams** are a useful way to visualize the variability information (Figure 3 right). Bar diagrams are a visualization of the occurrence matrices. *n+1* bars are created, one for each occurrence matrix. The length of each bar equals the number of rows in the respective matrix. Each bar is divided into three parts, with the lengths according to the number of core, shared and unique elements in each matrix. Bar diagram provides a quick overview over the amount of lines falling into each variability category, as well as over the sizes of variants relative to each other and to the total size of the union of all variants.
- **Support for more variants** is straightforward. Occurrence matrices and bar diagrams can be constructed for any number of compared variants, using the construction principles described above.
- **Subset calculations** can be run on the matrices, and their results can be visualized as additional colored bars in the bar diagram (Figure 4) or by highlighting the source lines in a text file. This dynamic visualization can be particularly useful for analyzing a larger number of variants and for complex calculations that involve many subsets of a Venn diagram. A calculation can for example return all lines in each variant that are shared with a given variant *A*, or all lines that belong to a set intersection such as $A \cap B$. The *shared* asset category can also be explored in more detail, for example by retrieving lines shared by exactly *k* variants. The calculations can be easily combined with each other (e.g. lines shared by exactly *k* variants, where one of the variants must be the variant *A*).
- **Computation time** of calculations can be held low by encoding each of the matrix rows as a bit vector. Then, each calculation can be run as a combination of bitwise AND, XOR and NOT operations performed on the matrix rows and on "calculation vectors" encoding the specific calculation. An int32 holds enough space to store a single bit-encoded matrix row, so the memory use for the matrices is also low.

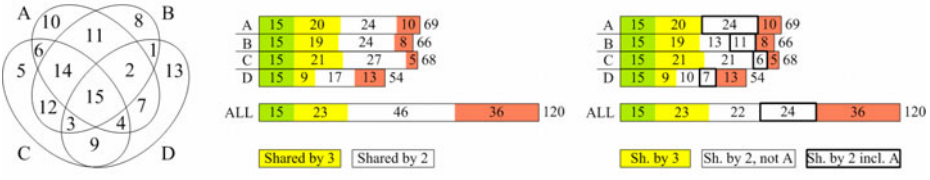


Fig. 4. Venn diagram for four example element sets (*left*). Bar diagram for *shared by 3* and *shared by 2* calculations performed on these sets (*middle*). Bar diagram with a further calculation: *shared by 2* variants, where one of these variants is *A* (*right*).

3 Generalization of the Technique

The demonstrated technique can be applied not only for diff results. Generally, the input data for occurrence matrices can be produced by a comparison operation performed on any kind of ordered or unordered lists of arbitrary comparable elements, such as strings, methods, classes, model elements, or other artifacts. The following conditions need to be met to construct the occurrence matrices in the general case:

- **Equivalence relation** on the compared elements needs to be defined. In the n-diff example, the relation was defined as string equality. In the context of product lines, the semantics of the equivalence relation is "can be treated as the same asset". Note that an equivalence relation has to be reflexive, symmetric and transitive.
- **Unambiguous assignment of equivalent elements across variants** needs to be assured. In case one or more elements from variant *B* are equivalent to the given element of variant *A*, there needs to be a clear rule, specifying whether an assignment can be made and if yes, which one of the potential matches should be used. In the n-diff example, this selection was defined by the ordering of the lines in the compared files and by the longest common sequence algorithm used by diff.

Example. Occurrence matrices and bar diagrams can be used to analyze variability in the sets of features of product line members: features can be treated as comparable elements, with the equivalence relation defined as "is the same feature". Since each feature exists only once in a given product, and the set of features can be treated as unordered, the unambiguous assignment of elements across variants is also assured.

4 Conclusions and Future Work

We proposed occurrence matrices and bar diagrams as a technique facilitating organizing, visualizing and processing of variability information. The proposed technique is well-suited for supporting detailed variability analysis in the context of reverse engineering and extractive product line adoption. As an example of using the technique, we demonstrated an n-ary text diff.

In the ongoing work, we plan to use the technique on other types of system representations and comparison functions (e.g. model-based), and to integrate it in a system structure browsing tool in order to enable variability analysis on all levels of the system structure hierarchy (source files, folders, subsystems and whole systems). We also plan to validate the technique by applying it to industrial software systems.

References

1. Faust, D., Verhoef, C.: Software Product Line Migration and Deployment. Softw. Pract. Exper. 33, 933–955 (2003)
2. Krueger, C.: Easing the Transition to Software Mass Customization. In: van der Linden, F.J. (ed.) PFE 2002. LNCS, vol. 2290, pp. 282–293. Springer, Heidelberg (2002)
3. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
4. Mende, T., Beckwermert, F., Koschke, R., Meier, G.: Supporting the Grow-and-Prune Model in Software Product Line Evolution Using Clone Detection. In: 12th European Conference on Software Maintenance and Reengineering (2007)
5. Duszynski, S., Knodel, J., Naab, M., Hein, D., Schitter, C.: Variant Comparison – A Technique for Visualizing Software Variants. In: 15th Work. Conf. on Reverse Eng. (2008)
6. Venn, J.: On the Diagrammatic and Mechanical Representation of Propositions and Reasonings. Philosophical Magazine Series 5, 10(59), 1–18 (1880)
7. Schmid, K.: A Comprehensive Product Line Scoping Approach and Its Validation. In: 24th International Conference on Software Engineering, pp. 593–603 (2002)

Recent Experiences with Software Product Lines in the US Department of Defense

Lawrence G. Jones and Linda Northrop

Software Engineering Institute
Pittsburgh, PA USA
lgj@sei.cmu.edu, lmn@sei.cmu.edu

Abstract. Since 1998, the Software Engineering Institute has held a series of annual software product line workshops with participants from the US Department of Defense (DoD) community. Over the years, the emphasis in the workshops has shifted from asking, "How can we do software product lines?" to sharing, "This is how we are doing software product lines." Both defense acquisition offices and their suppliers reported particularly impressive results in the 2010 workshop. This paper summarizes these recent workshop results, including descriptions of DoD product line efforts, implementation issues, and lessons learned.

Keywords: Software Product Lines, Software Product Line Adoption.

1 Introduction

In 1998, the Software Engineering Institute (SEI) began a series of annual product line practice workshops for the US Department of Defense (DoD)[1]. For the first few years, the workshop theme was *Product Lines: Bridging the Gap—Commercial Success to DoD Practice.* This theme demonstrated the main problem at the time: How do you transition proven product line practices into widespread use in the DoD? Over the years the workshop format and sponsorship have varied, but one thing is clear: product line practice is taking hold in the DoD. Early workshop discussions focused on *why* product line practice was difficult in the DoD; recent workshops have focused on *how* it is being done.

This paper summarizes four featured experience presentations from the 2010 SEI Software Product Line Workshop sponsored by the US Army and held in Orlando, Florida. For each we provide a description of the application, some implementation considerations, and key lessons learned. We conclude with some overall themes and observations.

2 Experience Summaries

2.1 Common Driver Training Product Line

Using simulators to train Army vehicle drivers is an attractive alternative to training on the actual vehicles, which have become increasing complex and expensive to operate.

[1] All workshop reports may be found at
http://www.sei.cmu.edu/productlines/start/dodplpworkshop/

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 486–490, 2010.

In 2004 the US Army Program Executive Office for Simulation, Training, and Instrumentation (PEO STRI) identified a need to develop a common line of driver training simulators for a range of ground vehicles. The goal was to exploit the commonality of training requirements to create common simulator elements, such as the instructor/operator station, motion base, image projectors, and data bases, while factoring out variant items, such as the vehicle cab, dashboard, and vehicle dynamics, as program-specific elements. Interchangeable vehicle cabs and dashboards coupled with vehicle-specific motion cues and realistic terrain databases provide a range of training scenarios.

The resulting Common Driver Trainer (CDT) product line provides the ability to create 80% of a new driver trainer from the CDT common elements. Use of CDT facilitates the rapid fielding of trainers for a range of vehicles, including the Abrams tank, the Stryker light armored vehicle, and the Mine-Resistant Ambush Protection (MRAP) vehicle.

Fielding CDT as a product line has yielded benefits in these areas:

1. *Requirements*. There is a common system requirements documents for all trainers. Vehicle-specific variants are covered in appendices.
2. *Time to field*. A training simulator can be put in the field very quickly – it literally takes more time to "bend the metal" than reuse the software. Use of CDT enabled meeting the aggressive MRAP simulator schedule: 120 days from contract award to first simulator delivery.
3. *Component reuse*. Eighty percent of a trainer comes from CDT common elements. The biggest factor in vehicle-specific variants is the difference between tracked and wheeled vehicles.

Lessons learned from the effort include:

- *Configuration Management*. CM was seen as the number one issue for any product line. Worse yet, CM tools do not provide adequate support for software product lines.
- *Personnel*. Personnel changes in government positions have a great impact on product line programs: There is a relatively small pool of people with product line experience to draw upon when filling vacancies and newcomers face a significant learning path.
- *Information Assurance*. Current regulations and guidance for information assurance don't even mention product lines. Until the people who write regulation and guidance documents become knowledgeable about product lines, compliance will be a struggle.

2.2 The Joint Fires Product Line

A military fire control system supports an observer in directing artillery fire and close air support. Developed under the auspices of PEO STRI, the Joint Fires Product Line (JFPL) is a set of training systems providing fire control virtual training for cross-service (currently special operations forces) observers and built from legacy systems.

Across the different products, variants include options for portable systems, classroom systems, and immersive trainers. Core assets include a system architecture (cov-

ering both hardware and software) and software modules. Non-software assets include guidance, support, and configuration documents, as well as user manuals and catalogs of products and software assets; savings in documentation was a key business motivation for the product line.

The actual government JFPL organization is small, with external teams responsible for product development. A "distribution agreement" complemented with a configuration management and quality assurance process governs the transfer of core assets to product developers and the acceptance of products and additional core assets back into the product line.

Lessons learned from the effort include:

- Don't underestimate the power of a cost-benefit analysis. If the effort is properly managed, the benefits far exceed the costs.
- Change management and collective vision are essential, as are champions with appropriate rank and influence to handle political roadblocks.
- Shared core assets require special testing processes.
- An early diagnostic is very helpful; the SEI's Product Line Technical Probe[2] was valuable.

2.3 Marine Corps Instrumentation Training System (MC-ITS)

The Army's Live Training Transformation (LT2) product line, developed by PEO STRI, is a family of systems supporting Army live training environments. The live training domain is also a responsibility of the U.S. Marine Corps (USMC) Range Modernization/Transformation (RM/T) program. The RM/T program office conducted an analysis and determined that they could leverage the capabilities of the Army's LT2 product line with over 80% direct mapping of requirements.

The first MCS-ITS increment was able to leverage the Army's Homestation Instrumentation System (HITS) (a member of the LT2 product line) to achieve 87% as-is reuse of common LT2 components. The remaining 13% are a combination of modified LT2 components and new Marine-Corps-specific components. Reusing LT2 components reduced the time to field the MCS-ITS increment by an estimated six-to-twelve months, and the reuse strategy allows two additional software development increments at no additional cost. There are also cost savings associated with integration and test, user training, and sustainability and maintenance. Other benefits identified included: the software is fully government-owned, allowing further sharing and avoiding intellectual property issues; improved exercise planning tools, and reduced problems of interfaces among joint applications.

2.4 Common Link Integration Processing Product Line

The Common Link Integration Processing (CLIP) is a software tactical data link (TDL) message processor that can be hosted on multiple platforms. Currently, CLIP is targeted to support B-1 and B-52 bombers, and F-35 fighters. The purpose of CLIP is to reduce cost (ownership, maintenance, and upgrade/refresh), to improve interop-

[2] Product Line Technical Probe is a service mark of Carnegie Mellon University.

erability by providing common implementations of TDL standards, and to eliminate stovepipe systems

Northrop Grumman, the developer of CLIP, was motivated to propose a product line approach because of the wording of the CLIP request for proposal (RFP) that referenced a "family of systems." However, since the RFP and statement of work (SOW) did not require a product line approach, the contract paid little attention to many product line practice aspects—there was no requirement for important contract deliverables such as a product line concept of operations, a product line practice description document, or a product line production plan.

Even so, the CLIP program achieved remarkable results because of their product line approach. The requirement to be able to use different data links for different missions meant that there were over 41,000 configuration parameters to handle, mainly because of the many message types. CLIP is able to handle all message formats, and changes to message formats, without requiring changes to the host platform software. Moreover, any required customization can be done on either the development side or the platform integration side. As a result, Northrop Grumman was able to build a CLIP system for another platform with just 10% of the effort that would previously be required while achieving 94% reuse of existing code. While these are impressive results, a lot of wasted effort could have been prevented by an RFP that explicitly specified a product line approach.

The lessons learned from the CLIP effort to date include:

- Both the government and the contractors must be ready to pursue product line engineering practices.
- Product line engineering must be built into the RFP, SOW, contract data requirements lists, and the system requirements document.
- The cost and schedule implications of a product line approach must be considered up front.
- Product line artifacts must be carefully managed by both the government and the contractors; traceability must be maintained throughout the development artifacts and documents.
- A product line approach requires both business and engineering buy-in.

3 Conclusions

Some common themes emerged during the workshop.

- Strategic reuse yielded benefits of reduced cost, reduced schedule, improved quality, and improved interoperability among systems.
- While a supplier can proceed without government support, a product line approach is much more effective if the government acquirer understands the approach and constructs their acquisition strategy and documents accordingly.
- Information assurance (IA) certification is becoming a real bottleneck in releasing systems. Work is needed to connect product line practice and IA so that the IA process can be streamlined to take advantage of the fact that a product within a product line can have a great deal of "pre-certification" built in.

- Configuration Management continues to be a challenge; it is difficult to keep multiple products in parallel development under configuration management
- Champions and executive sponsorship are critical.
- An architecture-centric approach is central to a product line effort; all programs explicitly used architecture methods from the SEI and others.
- Robust feedback loops between core assets and products were established.
- A cost-benefit analysis for the product line is important.
- A funding model that provides life cycle support to the product line is necessary.
- There is a need for coordination, communication, and education across all stakeholders (the ecosystem perspective).
- There is a need for more support for product line approaches from higher levels within the Army/DoD (e.g., acquisition guidance documents, information assurance requirements).

The reader will notice that two of the presentations were from PEO STRI programs and a third had ties to a PEO STRI effort. This is no accident. Product line thinking is endorsed by Dr. James T. Blake, the Program Executive Officer himself. Dr. Blake has said

> *The current operational environment requires quickly adaptable systems ... To better support this challenging environment, PEO STRI is adopting the deliberate and disciplined tenets of Product Line Acquisition to maximize adaptability.*

This kind of leadership support for product lines breeds success irrespective of the environment. The three successful product lines in PEO STRI are evidence of what is possible. The CLIP experience is evidence that defense contractors are attracted to (and can be successful with) software product lines even without DoD direction. These four experiences and others from other recent workshops in the series prove the viability and the benefits of software product lines within the US DoD.

Leviathan: SPL Support on Filesystem Level

Wanja Hofer[1], Christoph Elsner[1,2], Frank Blendinger[1],
Wolfgang Schröder-Preikschat[1], and Daniel Lohmann[1]

[1] Friedrich–Alexander University Erlangen–Nuremberg, Germany
[2] Siemens Corporate Research & Technologies, Erlangen, Germany
{hofer,elsner,wosch,lohmann}@cs.fau.de

A lot of configurable software, especially in the domain of embedded and operating systems, is configured with a source preprocessor, mostly to avoid run-time overhead. However, developers then have to face a myriad of preprocessor directives and the corresponding complexity in the source code, even when they might only be working on the implementation of a single feature at a time. Thus, it has long been recognized that tool support is needed to cope with this '#ifdef hell'. Current approaches, which assist the software developer by providing preprocessed views, are all bound to a special integrated development environment. This eliminates them from being used both in industry settings (where domain-specific toolchains are often mandated) and in open-source projects (where diverse sets of editors and tools are being used).

We therefore propose to tackle the problem at a lower level by providing variant views via a *virtual filesystem*. Our Leviathan filesystem allows the developer to mount one or more concrete product variants to work on. A variant is hereby defined as a set of enabled and disabled features (possibly output by a feature modeling tool to ensure correctness). A mounted variant essentially provides virtual files and directories, which are in fact slices of the original configurable code base, preprocessed by Leviathan's corresponding preprocessor component (e.g., CPP or M4). This technique enables the use of arbitrary file-based tools and tasks on that view. Operation includes read-only tasks such as reasoning about variants by viewing the differences between, or feeding them into syntax validators or code metric analysis tools. Furthermore, Leviathan's virtual files are also *editable* with arbitrary tools; Leviathan will merge back the changes into the configurable code base transparently in the background.

Leviathan's write-back support enables the developer to make changes directly on the mounted view. For instance, he can directly debug a mounted variant and modify the variant's code to get rid of a bug, eventually saving the changes in his editor. In the background, the write request will be handled by Leviathan's write-back engine, which is responsible for mapping *source configuration blocks* (enclosed in preprocessor directives) to *variant blocks* (actually visible in the view). This mapping and merging is either done heuristically (if the source configuration blocks are rather large in a given SPL) or with the help of markers, which Leviathan inserts as language-dependent comments in the mounted view to make the write-back process completely unambiguous.

By providing toolchain-independent views on preprocessor-configured code bases, Leviathan introduces SPL support in unprecedented domains like open-source projects (e.g., Linux) and industry projects.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, p. 491, 2010.

Introducing a Conceptual Model of Software Production

Ralf Carbon[1] and Dirk Muthig[2]

[1] Fraunhofer IESE,
Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
ralf.carbon@iese.fraunhofer.de
[2] Lufthansa Systems,
Am Prime Parc 2a, 65479 Raunheim, Germany
dirk.muthig@lhsystems.com

Abstract. Software development organizations today have to deliver products fast and tailored to the specific needs of customers. Software Product Line Engineering (PLE) has proven to support organizations in reducing time to market and increase the level of customization, but still software is not produced with similar efficiency than many hard goods today. We claim that organizations can improve the efficiency of software development if they apply a software production approach similar to other engineering disciplines. Key to successful software production is a product line architecture that is aligned with the production plan. As a first, step we introduce a conceptual model of software production that captures the relationship between the product line architecture and the production plan.

Keywords: Software production, Conceptual Model, Product Line Engineering, Product Line Architecture, Production Plan.

The product line architecture (PLA) is a core artifact of software production [1]. It defines, for instance, the architectural elements that make up a product line member. From the point of view of an architect, producing a product means to create, modify, remove, or assemble architectural elements. The production plan (PP) is another core artifact of software production. It describes the overall project approach, for instance, the iterations that are typically performed to incrementally product a product and the respective milestones. Furthermore, it defines the processes describing how the architectural elements are created modified, removed, or assembled in detail.

The conceptual model of software production elaborates on the relationships between conceptual elements of PLAs and PPs. Architectural elements, as one conceptual element of PLAs are related, for instance, to milestones. Typically, a certain number of architectural elements need to be touched to reach a milestone.

The knowledge captured in the conceptual model is supposed to be leveraged by product line architects and production planners to align PLAs and PPs. They should check, for instance, that not too many architectural elements need to be touched to reach a certain milestone, especially if such architectural elements are complex or if developers are not yet completely familiar with the used technologies. Hence, potential production problems, for instance, delays in reaching a certain milestone, can be detected early and appropriate solutions can be elaborated proactively.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 492–493, 2010.

The relevance of the relationship between architectures and project plans has already been mentioned in the literature, for instance, in [2]. Especially in PLE, where the production process will be repeated many times, an up-front investment in aligning PLA and PP to prevent production problems promises high returns on investment.

References

[1] Carbon, R.: Improving the Production Capability of Product Line Organizations by Architectural Design for Producibility. In: Proceedings of the SPLC 2008, vol. 2, pp. 369–375 (2008)

[2] Paulish, D.J.: Architecture-Centric Software Project Management. Addison-Wesley, Reading (2002)

Product Line Engineering in Enterprise Applications

Jingang Zhou[1,*], Yong Ji[2], Dazhe Zhao[1,2], and Xia Zhang[1,2]

[1] Northeastern University, 110819, Shenyang, China
[2] Neusoft Corporation, 110179, Shenyang, China
{zhou-jg,jiy,zhaodz,zhangx}@neusoft.com

Keywords: Enterprise application, enterprise application platform, software product line.

Software product line engineering (SPLE) has been gotten considerable adoption in software intensive domains. Enterprise applications, a long lived application domain for enterprise operations with IT, are mainly developed as single system engineering. It is hard to adopt SPLE in enterprise application domain (EAD) with so much diversity, plus the complex and fast evolving technologies.

Neusoft provides application solutions for the fields of e-government, education, finical, telecom, traffic, electric power, enterprise, and so on. To deliver products efficiently and keep competitive, we introduce SPLE and have developed a common technical platform, called *UniEAP*^{TM} for enterprise solutions, which contains application frameworks, as well as many technical components (e.g., workflow, rule engine, report, cache, etc.). Due to the diversity in EAD, some derived platforms have been developed to include more general business components in fields like social insurance, electric power, and telecom to facilitate application development in these sub-domains. By this hierarchical domain engineering (DE) units organization [1], application engineering (AE) teams have successfully constructed more than two hundreds of solutions annually for different business lines in Neusoft.

Technical and organization management plays an important role to collaborate the decentralized AE and DE teams. In such a context, periodic workshops are important to platform features timeline schedule and training. In a fast evolving domain, it is useful to layer assets by reusability and pay more attention to those with high reusability. Black/white box components, frameworks, application templates, partial code generators and even code snippets are all enabling strategies. The synthesized approach makes SPLE in EAD showing characteristics in higher maturity levels [1] than just platform. For assets assembly, OSGi bundles can be used as components packaging format to realize modularity and features metadata can be defined in MANIFEST.MF to easing mapping between requirements and software artifacts.

One of the most difficulties is assets evolution in which multi-version control, dependency management, requirements/features traceability, all need consideration and tools support. Thus it is key to constraint the number of variation points, e.g., by specifications, however, flexibility and configurability must be taken into account.

Reference

1. Bosch, J.: Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 257–271. Springer, Heidelberg (2002)

* Corresponding author. Sponsored by the State 863 High-Tech Program (2008AA01Z128).

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, p. 494, 2010.

Case Study of Software Product Line Engineering in Insurance Product

Jeong Ah Kim

Department of Compter Education, Kwandong Univeristy,
522 NaeKok Dong, KangNung, KangWon, 210-701, Korea
clara@kd.ac.kr

SPLE is a process to analyze a set of products and to identify the reusable components in the product family. And these components can be used for developing a new product in the product domain. Insurance sales system is software intensive system and the competitive power of software system determines the growth rate of business in insurance market. Sale system of insurance product is main target for improving their competition. Since the number of new insurance product is increasing and time-to-market is getting shorten, the needs for software product line engineering adoption is increased. As new insurance products are released, small-sized projects should be launched to add new software insurance products into existing information system. Since a software product module was implemented for each insurance product, there are so many redundant parts among the software insurance modules. Redundant codes make the problems in updating the already registered insurance products and make the cost of maintenance increasing. Software product line engineering can be the solution for redundancies in software systems and for reducing the cost of maintenance. To adopt the software product line to insurance sales system, 2 areas can be candidates (1) insurance sales system, (2) software insurance products. To smooth the adaptation, software insurance products is the first target for SPLE.

Variability in software insurance products is classified as (1) variability of attribute, (2) variability of the domain of attribute (3) variability of attribute value, (4) variability in computation logic, (5) variability of condition. (6) variability in work flows. Result of domain engineering was described as feature model to explain what features are common and what features are variant points. Variation mechanism with XML-based ontology modeling was introduced. XML-based variability specification can be flexible way to extend the scope of variability without affecting other modules. With XML, if existing modules are not interested in new added attributes, then they can ignore new attributes. New added attribute and related computational logic and condition can affect to new insurance product which is the reason of variability. It means that fixed variability modeling is not required at the early stage of software insurance product SPLE. In XML schema, variation point is defined as tag with business meaning and is defined as rule expression name. In XML instance, variant is modeled as value and rule. Evolution of variability modeling is very important issue since new concepts and new attributes are increasingly added. XML-based variability specification make possible to add new features and to ignore existing features so that the version control of XML-based variability specification is important and traceability from features to application module should be managed.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, p. 495, 2010.

Using Composition Connectors to Support Software Asset Development

Perla Velasco Elizondo

Centre for Mathematical Research, (CIMAT). 36240 Guanajuato, Mexico
pvelasco@cimat.mx

Software Product Line (SPL) approaches enable the development of software product variants by *reusing* a set of *software core assets*. These assets could have variant features themselves that can be configured in different ways to provide *different behaviours*. Unfortunately, in many SPL approaches software core assets are constructed *from scratch* and in an *ad hoc* manner.

In previous work, it has been introduced a new approach to component composition within the context of Component-based Development [1]. In this approach, components are *passive* and they do not request services from other components. Rather, they execute their provided services only when invoked by connectors. Connectors are *exogenous* to components and they encapsulate well-known *communication schemes*, e.g. [2]. These connectors are indeed first-class compilation units that admit some sort of parametrisation to indicate the components they compose and the services that should be considered in these compositions.

In [3] a catalogue of these connectors is presented. Thus, this work shows the feasibility of utilising these connectors to generate software core assets. Specifically, the feasibility of generating a set of *reusable software core assets* for a Home Service Robots product line. The assets are generated by composing a set of passive technology-intensive components (e.g. speech and sound recognisers, obstacle detectors) into specific arrangements via our connectors. Although this piece of work is only at an initial stage, we realised that it has a set interesting features: (i) it provides a *systematic* and *consistent* means to generate software core assets -it allows reuse of a well-defined composition process, (ii) it *maximises reuse* –it promotes reuse of both components and connectors and (iii) it admits some level of *automation* –it mitigates software asset development effort.

We agree, however that a complete case study implementation must be carried out to better support all these claims. Similarly, a better integration of the artefacts describing the product line's variability with our approach is required. Thus, our future work will focus on these issues.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 496–497, 2010.

References

1. Lau, K.-K., Velasco Elizondo, P., Wang, Z.: Exogenous connectors for software components. In: Heineman, G.T., Crnkovic, I., Schmidt, H., Stafford, J., Szyperski, C., Wallnau, K. (eds.) Proceedings of 8th International SIGSOFT Symposium on Component-based Software Engineering, pp. 90–106. Springer, Heidelberg (2005)
2. Russell, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., Mulyar, N.: Workflow control-flow patterns: A revised view. Technical Report BPM-06-22, BPM Center (2006)
3. Velasco Elizondo, P., Lau, K.-K.: A catalogue of component connectors to support development with reuse. Journal of Systems and Software 83(7), 1165–1178 (2010)

Feature-to-Code Mapping in Two Large Product Lines

Thorsten Berger[1], Steven She[2], Rafael Lotufo[2],
Krzysztof Czarnecki[2], and Andrzej Wąsowski[3]

[1] University of Leipzig, Germany
berger@informatik.uni-leipzig.de
[2] University of Waterloo, Canada
{shshe,rlotufo,kczarnec}@gsd.uwaterloo.ca
[3] IT University of Copenhagen, Denmark
wasowski@itu.dk

Large software product lines have complex *build systems* that enable compiling the source code into different products that make up the product line. Unfortunately, the dependencies among the available build options, which we refer to as *features* and their mapping to the source code they control, are implicit in complex imperative build-related logic. As a result, reasoning about dependencies is difficult; this not only makes maintenance of the variability harder, but also hinders development of support tools such as feature-oriented traceability support, debuggers for variability models, variability-aware code analyzers, or test schedulers for the product line. Thus, we advocate the use of explicit *variability models*, consisting of a *feature model* specifying the available features and their dependencies and a *mapping* between product specifications conforming to the feature model and the implementation assets.

Previously, we extracted feature models from the Linux kernel and the Ecos embedded operating system. The Ecos model directly embeds the feature-to-code mapping. However, this is not the case for Linux. Now, we extract the feature-to-code mapping from the build systems of the prominent operating systems Linux and FreeBSD.

We represent the mapping as *presence conditions*. A presence condition (PC) is an expression on an implementation artifact written in terms of features. If a PC evaluates to *true* for a configuration, then the corresponding artifact is included in the product.

We show that the extraction of mappings from build systems is feasible. We extracted 10,155 PCs, each controlling the inclusion of a source file in a build. The PCs reference the total of 4,774 features and affect about 8M lines of code. We publish[1] the PCs for Linux and FreeBSD as they constitute a highly realistic benchmark for researchers and tool designers.

[1] http://code.google.com/p/variability

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 498–499, 2010.

In the poster, we describe the build systems of the Linux and FreeBSD kernel as well as our approach to transforming the imperative build-system logic into a large Abstract Syntax Tree (AST) and to deriving presence conditions. Furthermore, we expand on basic characteristics of the resulting expressions. We hope our work deepens understanding of variability in build systems and that the insights will eventually lead to extracting complete variability models encompassing the feature model and the mapping from features to code.

The Rise and Fall of Product Line Architectures

Isabel John[1], Christa Schwanninger[2], and Eduardo Almeida[3]

[1] Fraunhofer Institute for Experimental Software Engineering (IESE)
Isabel.John@iese.fraunhofer.de
[2] Siemens AG
christa.schwanninger@siemens.com
[3] Federal University of Bahia (UFBA) and RiSE
esa@dcc.ufba.br, esa@rise.com.br

Abstract. This panel addresses questions around architecture like: How do you think a good product line architecture should look like? How much up-front design do we need for a product line architecture? What are hot research topics in product line architecture? The panel is organized as a goldfish bowl, where the panelists are in the middle of the audience and panelists change during the panel.

1 Motivation

Although architecture has been recognized as **the** central artifact in product line engineering since many years, it doesn't seem to be the hot topic anymore. There was no session on architecture at last SPLC, there was only one tutorial mentioning architecture in the title. So, is the concept of a product line architecture dead? Or is it just normal to have one, so we don't have to talk about it anymore? In this panel we want to have a look at these questions: Product line architectures might be irrelevant in the era of agile processes and open source, but if they are not we want to revive this topic for the SPL community.

Questions to start the panel with could be:

- How do you think a good product line architecture should look like?
- How much up-front design do we need for a product line architecture?
- Is parallelizing Domain Engineering (DE) and Application Engineering (AE) possible without a stable product line architecture?
- How do DE and AE collaborate in evolving architecture? What are the extremes in the spectrum from DE controlled to collaborative?
- What are hot research topics in product line architecture?
- What is the relationship between architecture and scope?
- What are the experiences creating architecture from available assets and products?
- How to deal with varying quality attributes for a product line?
- Who has a product line architecture? How does it look like? Is it easy to handle?
- And what to do with the architecture when it comes to application engineering?
- Do we really need a dedicated product line architecture?

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 500–501, 2010.

- Do we need research on product line architectures?
- And how do the variabilities in the architecture look like (if there are any)?

2 Panel Format

We chose a format called "Goldfish bowl" instead of the typical panel format. Goldfish bowls provide broad-ranging discussion on a general topic. There already was a Goldfish bowl panel on PL business return a SPLC 2009 [1], which was a great success. We largely will follow the organization and format that we had last year.

There are several variants of a goldfish bowl. We roughly follow the variant described in [2]:

A goldfish bowl is a bit like a panel discussion, but the people on the panel change during the discussion. The rules are:

- start with a small circle or semi-circle of about 5 chairs in the middle of the room;
- arrange all the other chairs facing towards these, so you have several concentric circles;
- the goldfish bowl has a topic which is usually quite broad, and the organizer usually asks two or three people to get the discussion started and explains the rules to the audience;
- during the discussion, anyone sitting on one of the central chairs can speak, but no-one else can;
- if someone wants to speak, they have to sit on one of the central chairs, even if it's just to ask a question;
- one of the central chairs is always kept free for this purpose;
- whenever all of the central chairs are occupied, at least one person has to leave to create a new vacant chair.

References

[1] http://www.sei.cmu.edu/splc2009/files/Goldfish_Bowl_Panel_SPLC_2009-A3.pdf
[2] http://www.xpday.org/session_formats/goldfish_bowl

The Many Paths to Quality Core Assets

John D. McGregor

Clemson University
johnmc@cs.clemson.edu

Can all of the different approaches to software product line operation lead to quality core assets? Some organizations use a recognized method such as the Software Engineering Institute's Framework for Product Line Practice [4] while others use a homegrown variant of their non-product line process. Some organizations iteratively mature their core assets while others build proactively.

Is there value in quality? The Craftsman approach to software development believes there is and takes extra time to ensure high levels of quality [5]. Others view quality like any other requirement. Satisfying the requirement by whatever small amount is sufficient.

Should discussions of quality in a product line address separate quality factors for core assets and product-specific components? Product line organizations often issue reports of reduced defect density rates that do not distinguish between the defect densities of core assets and product-specific components. Time spent on core assets has a multiplier effect by exporting that improved defect count to a large percentage of products [1].

How do core assets reach maturity most rapidly? Some organizations plan a migration path for each asset [2], some scratch their collective heads when a late change in the core assets suddenly becomes necessary, and others keep a loose, iterative approach that allows flexibility.

How are quality requirements different in a software product line? In a software product line, everything can be variable including quality [3]. That is, different products may have very different requirements for specific quality attributes.

This panel session will present diverse views on how to achieve quality products in a software product line organization. We will aggressively seek audience participation in an effort to answer these questions with actual experience.

References

1. In, H.P., Baik, J., Kim, S., Yang, Y., Boehm, B.: A quality-based cost estimation model for the product lin life cycle. Communications of the ACM 49(12) (2006)
2. McGregor, J.D.: The evolution of product line assets. Tech. rep., Software Engineering Institute (2003)
3. Myllärniemi, V., Männistö, T., Raatikainen, M.: Quality attribute variability within a software product family architecture. In: Proceedings of the Conference on the Quality of Software Architectures (2006)
4. SEI: Framework for software product line practice. Tech. rep., Software Engineering Institute (2010)
5. Völter, M., McGregor, J.D.: The model craftsman. Executive Update (2010)

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, p. 502, 2010.

Pragmatic Strategies for Variability Management in Product Lines in Small- to Medium-Size Companies

Stan Jarzabek

School of Computing, National Univeristy of Singapore, Singapore
stan@comp.nus.edu.sg

Most SPLs in small- to medium-size companies evolve from a single successful product. Initially, each new product variant is often developed by ad hoc reuse - copy and modify - of source code files implementing existing products. Versions of source files pertinent to different products are stored under a Software Configuration Management (SCM) tool such as CVS or SVN. As the number of customers and relevant product variants increases, such ad hoc reuse shows its limits: The product size grows as we implement new features in response to customer requests. At the same time, we need maintain all the released product variants, so we have more and more code to maintain. Also with a growing customer base (good for our business!), increasing product variability becomes a challenge for ad hoc reuse: How do we know which versions of source files should be selected from SCM repository for reuse in development of a new product? How should we customize them and then integrate to build a new product? These problems may become taxing on company resources.

We can already start observing initial symptoms of the above problems as the number of product variants reaches 4-5. As maintaining product variants and implementing new ones become more and more time-consuming, managing variability using systematic software Product Line (SPL) techniques can help a company sustain the business growth.

Setting up and stabilizing a common architecture for SPL product variants is the first step. Some variant features of products can be nicely mapped into architectural components. Handling such features becomes easy with plug-in components.

Component-based approaches and platforms provide effective reuse solutions at the middleware level. However, in application domain-specific areas such as business logic or user interface, despite many similarities, software is still developed very much from scratch. One of the reasons why this happens is that variability affecting middleware components is lesser and can be easier localized than variability affecting upper system layers.

In most application domains, plug-in component technique is not enough. The impact of variant features cannot be contained at the component level, but spreads freely across many components, affecting their code at many variation points. To manage such "troublesome" variability in core assets, companies typically adopt variation mechanisms such as preprocessing, manually commenting out variant feature code, parameter configuration files, Ant, or annotations (Java/JEE). Such variation mechanisms are simple and available for free. Most developers can understand them without training.

The need for multiple variation mechanisms arises because we face many different variability problems when designing core assets of an SPL: Some variant features require customizations at the file level, others trigger many customizations in core

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 503–504, 2010.

asset code, yet others may involve a combination of file- and code-level customizations. Each variation mechanism is meant to handle only one type of variability situation, and sometimes we must use variation mechanisms together to handle a give variability situation.

The above strategy to building an SPL is cost-effective and works well, as long as the number of variant features differentiating products is below 50, and the product size is below 50 KLOC.

As our SPL grows, problems usually emerge: Features get complicated; One variant feature may be mapped to many variation points, in many components, and it is difficult to figure out to which ones and how; Features often are inter-dependent, and inclusion of one feature into a custom product must be properly coordinated with modifications of yet other features; Core reusable assets become heavily instrumented with variation points, and using multiple techniques to manage variability makes the core assets even more complex to work with.

If the above picture reflects your experience, you may find this tutorial useful. We review techniques commonly employed for SPL variability management at architecture and the detailed code level. This part of the tutorial gives a balanced view on how far they can lead you, and complications that typically arise in time. If you experience the pains already, you will better know their sources.

In the second part of the tutorial, we examine XVCL (XML-based Variant Configuration Language) variation mechanism [1][2] that exercises the total control over SPL variability, from architecture, to component configuration, to any detail of code (e.g., variations at the source statement, expression or keyword level). XVCL streamlines and automates customizations involved in implementation of selected variant features into custom products, from component re-configuration, to detailed customizations of component code. The approach replaces the need for multiple variation mechanisms, and avoids the problems of digging out feature customization and reuse information from SCM repositories. It complements conventional architecture-centric, component based design for reuse, and works with any conventional programming language and/or platform such as JEE, .NET, Ruby on Rails or PHP.

In the tutorial, we discuss two industrial case studies of small- to medium-size SPLs with XVCL: We review histories of Web Portals by ST Electronics (Info-Software Systems) Pte Ltd in Singapore, and Wingsoft Financial Management Systems by Fudan Wingsoft Ltd., a software company in China. We describe the strengths and weaknesses of managing variability using advanced conventional techniques, migration into XVCL-based SPLs, and costs/benefits of managing SPLs with XVCL.

References

[1] Jarzabek, S.: Effective Software Maintenance and Evolution: Reuse-based Approach. Taylor & Francis, CRC Press (2007)

[2] XVCL (XML-based Variant Configuration Language) method and tool for managing software changes during evolution and reuse, `http://xvcl.comp.nus.edu.sg`

Building Reusable Testing Assets for a Software Product Line

John D. McGregor

Clemson University
johnmc@cs.clemson.edu

Abstract. A software product line presents all of the usual testing challenges and adds additional obligations. While it has always been important to test the earliest products of development, a product line makes this much more important in order to avoid a combinatorial explosion of test suites. We will consider ways to reuse test cases, test infrastructure, and test results. We will consider fault models for software product lines and then develop a set of test plans based on those fault models.

Testing consumes a significant percentage of the resources required to produce software intensive products. Data from one company, reported at SPLC 2009, indicated that 31% of the product line organizations effort was required for component test, integration, and validation while only 10% was required to develop the code. This obviously includes the testing effort for the product line and suggests room for considerable savings. The exact impact on the project is often hard to evaluate because testing activities are distributed over the entire scope of the development effort as it is in any product development effort. In fact few product line organizations have a coordinated plan for all of the testing activities. Those test activities conducted by developers, integration teams, and system testers often operate in separate compartments with little or no communication with each other. In this tutorial we take a comprehensive end-to-end view of the testing activities and roles that should be present in a software product line organization. We consider testing activities that begin during product conception and that continue after deployment to customers.

Testing "a product line" is like testing several products, but since there are many dependencies among the products, we can plan and execute the testing activities as a group and realize significant savings. The commonality and variability analysis performed early in the life of the product line provides valuable information for testing. The structure of the test cases should approximate that of the products and the variation points in the product line architecture give information for identifying separate test fragments. [2]

The Software Engineering Institute (SEI) identifies three areas of responsibility in a product line organization we relate these to testing:

- Organizational managers have responsibility for establishing the test strategy for the organization in general and the product line in particular. These activities are directly related to the business goals and scope of the product line.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 505–506, 2010.

- Technical managers have responsibility for planning the numerous test activities needed to implement the test strategy. These activities are planned in concert with the development activities to coordinate milestones and resources.
- Software engineers have responsibility for implementing the planned activities. They select the specific test cases necessary to achieve specific test coverage levels and implement any software needed to apply the test cases to the software under test.

A fault model describes the types of faults that might be present in a given environment. The fault model for a software product line covers a wide range of fault types. In this tutorial we will examine several representative faults and use them to motivate the testing technique we employ. These give a basic starting point but each organization must construct their own fault model that works with their development process.

The close relationship between developing software and testing it results in the test activities being crafted with knowledge of the chosen development process. The method engineer arranges the testing activities so that they are timely and have the appropriate perspective for their position in the development process. This tutorial considers test techniques and test process models.

The content of the tutorial is based on the software product line literature, the applied research conducted at the Software Engineering Institute, and the technical consulting of the SEI. [3] [1]

References

1. Kazman, R., Klein, M., Clements, P.: Atam: Method for architecture evaluation. Tech. Rep. CMU/SEI-2000-TR-004, Software Engineering Institute (2000)
2. Knauber, P., Schneider, J.: Tracing variability from implementation to test using aspect-oriented programming. In: Proceedings of the Software Product Line Conference 2004 (2004)
3. McGregor, J.D.: Testing a software product line. Tech. Rep. CMU/SEI-2001-TR-022, Software Engineering Institute (2001)

Production Planning in a Software Product Line Organization

John D. McGregor

Clemson University
johnmc@cs.clemson.edu

Abstract. Production planning gives early guidance concerning how products should be built and hence how core assets should be designed. The production strategy addresses business goals through product building. The production method implements the production strategy by delineating exactly how a product is built from core assets. The production plan instantiates the production method for a specific product. In this tutorial we will layout production planning in a software product line and provide examples from a number of different product lines.

Most software product line organizations recognize the need for two roles: core asset developers and product builders. These roles may both be assumed by an individual or each may be assumed by persons who are in different administrative units, in different geographic locations, or of vastly different skill levels. For example, a corporation may have one lab assigned to produce core assets and other labs around the world to use those assets to produce products. The greater the separation among these people the greater the need for communication and coordination regarding product production.

Production planning is used in many industries to coordinate the efforts of external suppliers who supply parts and to structure the assembly line where products are produced. The need for coordination in a software product line organization is even greater than in hard goods manufacturing because product production is less constrained by physical properties or industrial standards. Our research has shown that organizations that fail to plan production are more likely to fail than those that do plan. The goal of this tutorial is to provide participants with techniques for conducting production planning.

Production planning takes place in the context of an ecosystem that encompasses the software product line. It consists of the vendors who supply parts of the product and the different buyers of the product. Examining the ecosystem gives the production strategist information about competition in the market and about available resources.

Production planning begins with development of a production strategy that ties the business goals of the product line organization to the means of production. The strategy specifies an approach that addresses the forces posed by existing competitors, potential entrants into the same markets, existing products that could be substituted, suppliers of resources, and purchasers of products. The

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 507–508, 2010.

production strategy identifies the types and amounts of resources that will be devoted to product production and how these allocations will meet specific goals.

The production strategy guides the creation of an implementation approach which includes the method of production and specific production plans. The production method implements the production strategy by defining process models, modeling conventions, and technologies that will be used to produce core assets and products. Techniques from method engineering are used to define the production method. Each product team can specialize the product line method to fit the specific circumstances of the product or customer.

The production method specifies the form of the attached process that is suitable for building a production plan. The attached process for a core asset provides detailed instructions for using the asset in building a product. Each core asset has an attached process. As the variation points are resolved, the attached processes for the assets used at the variation points are added to the other attached processes to define the method specific to that product.

The production strategy and method are instantiated in a production plan. The plan provides the details of how products will be produced, when they will be produced, and by whom they will be produced. The variety of products in a product line often requires a variety of production techniques. Each core asset used in production is accompanied by an attached process that is integrated into the production process that guides product production.

The content of this tutorial is based on the applied research and technical consulting of the Software Engineering Institute. [4] [2] [3] [1]

References

1. Chastek, G., Donohoe, P., McGregor, J.D.: Product line production planning for the home integration system example. Tech. Rep. CMU/SEI-2002-TN-009, Software Engineering Institute (2002)
2. Chastek, G., Donohoe, P., McGregor, J.D.: A study of product production in software product lines. Tech. Rep. CMU/SEI-2004-TN-012, Software Engineering Institute (2004)
3. Chastek, G., Donohoe, P., McGregor, J.D.: Formulation of a production strategy for a software product line. Tech. Rep. CMU/SEI-2009-TN-025, Software Engineering Institute (2009)
4. Chastek, G., McGregor, J.D.: Guidelines for developing a product line production plan. Tech. Rep. CMU/SEI-2002-TR-006, Software Engineering Institute (2002)

Transforming Legacy Systems into Software Product Lines

Danilo Beuche

Pure-systems GmbH, Agnetenstr. 14, 39106 Magdeburg, Germany
danilo.beuche@pure-systems.com

Abstract. This tutorial discusses many aspects of the migration process in an organization when existing software systems are used as starting point for a software product line. It is intended to provide food for thought as well as practical approaches for the migration.

1 Tutorial Motivation

Often organizations face the problem that after a while their software system is deployed in several variants and the need arises to migrate to systematic variability and variant management using a software product line approach.

In practice most organizations cannot afford to start a software product line development from scratch and therefore have to use as much existing software assets as possible. Discussion of (successful) transition techniques thus helps those organizations to decide for adoption of a software product line approach. Since the software product line development is still in a phase were widespread use has not been achieved the tutorial tries to help attendees to increase the number of SPL in the industry.

2 Tutorial Contents

The tutorial will discuss issues coming up during this migration process mainly on the technical level, but also discusses some of the organizational questions. The goal of the tutorial is to give attendees an initial idea how a transition into a software product line development process could be done with respect to the technical transition.

The tutorial starts with a brief introduction into software product line concepts, discussing terms such as problem and solution space, feature models, versions vs. variants.

Tutorial topics are how to choose adequate problem space modeling, the mining of problem space variability from existing artifacts such as requirements documents and software architecture. Also part of the discussion will be the need for separation of problem space from solution space and ways to realize it. A substantial part of the tutorial will be dedicated to variability detection and refactoring in the solution space of legacy systems.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 509–510, 2010.

3 Tutorial Audience

The intended audience is practitioners from industry (software product line novice to intermediate). Attendees should have a basic understanding software design; some knowledge about software product lines in general is helpful but not required.

Systems and Software Product Line Engineering with the SPL Lifecycle Framework

Charles W. Krueger

BigLever Software, Inc.
Austin, TX, USA

Abstract. Mainstream forces are driving Software Product Line (SPL) approaches to take a more holistic perspective that is deeply integrated into the systems and software engineering lifecycle. These forces illustrate that SPL challenges will not be solved at any one stage in the product engineering lifecycle, nor will they be solved in independent and disparate silos in each of the different stages of the lifecycle. We explore our response to these forces – a SPL Lifecycle Framework. The motivation for this technology framework is to ease the integration of tools, assets and processes across the full systems and software development lifecycle. The goal is to provide product line engineers with a common set of SPL concepts and constructs for all of their tools and assets, at every stage of the lifecycle, and to assure that product line development traceability and processes flow cleanly from one stage of the lifecycle to another.

1 Introduction

In this tutorial, we describe how SPL Lifecycle Framework has enabled successful large-scale deployments on highly recognizable systems in all three branches of the US Department of Defense and some of the largest companies in aerospace and defense, automotive manufacturing, alternative energy, industry automation, e-commerce, medical systems and computer systems, as well as three inductees in the SPLC Software Product Line Hall of Fame. We explore how the SPL Lifecycle Framework provides all product line engineers and managers – including systems analysts, requirements engineers, architects, modelers, developers, build engineers, document writers, configuration managers, test engineers, project managers, product marketers and so forth – with a common set of SPL concepts and constructs for all of their tools and assets, at every stage of the lifecycle, and to assure that product line development traceability and processes flow cleanly from one stage of the lifecycle to another. The key constructs covered are:

- A *feature model* that you use to express the feature diversity (optional and varying feature choices) among the products in your product line.
- A uniform *variation point* mechanism that is available directly within your tools and their associated assets, to manage feature-based variations in all stages of the engineering lifecycle.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 511–512, 2010.

- A *product configurator* you use to automatically assemble and configure your assets and their variation points – based on the feature selections you make in the feature model – to produce all of the assets and products in your product line.

A key capability of the SPL Lifecycle Framework is the integration of SPL concepts into existing tools, assets and processes across the systems and software development lifecycle, including many of the industry standards in programming languages and compilers, integrated development environments, requirements management, change and configuration management, build systems, quality management, model driven development, word processors and documentation. SPL concepts and constructs in the framework expand your collection of tools and processes – making them product line aware – in three dimensions of distinct and synchronous SPL concerns.

- *Multi-product*. The feature-based variation management and automated production line necessary to engineer and deliver the multiple products in a product line.
- *Multi-phase*. The tools and traceability necessary to support the multiple phases of a product line engineering lifecycle – from business case, to requirements, design, implementation, testing, delivery, maintenance and evolution.
- *Multi-baseline*. Change management and configuration management for a product line are done on multiple evolving baselines of the SPL assets, analogous to the supply chains for automotive manufacturing.

The final piece of the solution is a 3-tiered SPL Methodology. The methodology is a pragmatic new-generation SPL methodology with a practical tiered approach that allows you to make a non-disruptive transition and successfully operate your production line. Each tier builds upon and is enabled by the previous tier:

- *Base tier*: Implement variation management and an automated production line by leveraging existing tools and legacy assets.
- *Middle tier*: Organize teams for horizontal SPL asset focused development rather than vertical product-centric development.
- *Top tier*: Transition business from product-centric to feature-based portfolio management, where the portfolio evolves by adding or modifying common and varying features supported by a production line.

We will describe observations and firsthand experiences on how the SPL Lifecycle Framework has enabled mainstream organizations, with some of the largest, most sophisticated and complex, safety-critical systems ever built, to transition legacy and new systems and software assets to the SPL approach.

Managing Requirements in Product Lines

Danilo Beuche[1] and Isabel John[2]

[1] Pure-systems GmbH
danilo.beuche@pure-systems.com
[2] Fraunhofer Institute for Experimental Software Engineering (IESE)
Isabel.John@iese.fraunhofer.de

Abstract. Any organizations develop software or software –intensive products, which are can be seen as variants or members of a product line. Often the market demands variability and the software organization expects productivity benefits from reuse. In any case, complexity of the software development increases. Requirements management plays a central role in this, when it comes to mastering the complexity. In this tutorial we will give an overview on how to analyze, build and manage common and variable requirements for a product line.

1 Topic

The tutorial aims at providing the essential knowledge for successfully running requirements management for product lines and variant rich development scenarios. Besides explaining methods also information about implementing the methods with standard tools is given.

Special consideration is given to linking the presented methods into the practice using examples and case studies (e.g. [1][2]). Interaction with the participants is integral part of this tutorial.

In this tutorial we cover product line requirements management. Therefore this tutorial covers the following topics:

- The importance of product line requirements management, difference to traditional reuse strategies.
- Methods for managing requirements in a product line including discussion of requirements reuse granularity.
- An approach for extraction of commonality and variability from existing requirements documents (CAVE)[3].
- Embedding the requirements management activities into the overall organization in a product line development.

With these topics we cover technological, organizational and business aspects of requirements management for product line, enabling practitioners to start with requirements management for product lines on a solid basis. The intended audience is practitioners that want to learn how to carry out requirements management for product line successfully, as well as researchers that want to learn about state of the practice of product line requirements management.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 513–514, 2010.

2 Plan

The tutorial consists of three parts:

1. Introduction
    a. Product Line Basics
    b. Development Scenarios

2. Methods and Tools
    a. Variability Concepts for RE
    b. Feature Modelling
    c. Mapping to tools
    d. Requirements Variation Point Extraction

3. Process and Organization
    a. PL and PL RE Lifecycle
    b. Roles and Boards in PL
    c. Common Pitfalls

Both presenters have a year long experience in working with requirements engineering and management in real projects. The tutorial is based on their experience with product line engineering in many industrial projects and combines their viewpoints to a holistic view on the topic.

Acknowledgements

The work of Isabel John was partially supported by the Fraunhofer Innovation Cluster for Digital Commercial Vehicle Technology (DNT/CVT Cluster).

References

[1] Beuche, D., Birk, A., Dreier, H., Fleischmann, A., Galle, H., Heller, G., Janzen, D., John, I., Kolagari, R.T., von der Maßen, T., Wolfram, A.: Using Requirements Management Tools in Software Product Line Engineering: The State of the Practice. In: Proceedings of SPLC 2007, pp. 84–96 (2007)

[2] Birk, A., Heller, G., John, I., Schmid, K., von der Maßen, T., Müller, K.: Product Line Engineering: The State of the Practice. IEEE Software 20(6), 52–60 (2003)

[3] John, I.: Using Documentation for Product Line Scoping. IEEE Software 27(3), 42–47 (2010)

Evolutionary Product Line Scoping

Isabel John and Karina Villela

Fraunhofer Institute for Experimental Software Engineering (IESE),
Fraunhofer Platz 1, 67663 Kaiserslautern, Germany
{Isabel.John,karina.villela}@iese.fraunhofer.de

Abstract. Product Line Engineering has a widespread use in industry now. Therefore there is a high need for customizable, adaptable, and also for mature methods. Scoping is an integral part of Product Line Engineering. In this phase we determine where to reuse and what to reuse, establishing the basis for all technical, managerial, and investment decisions in the product line to come. In this tutorial we will give an introduction on how to analyze an environment with the purpose of planning a product line and its future evolution.

1 Introduction

This tutorial aims at giving an understanding of how to identify and analyze common and variable capabilities of systems, and describing how to integrate scoping and evolution planning in an architecture centric Product Line Engineering approach. Systematic identification and description of commonalities, variabilities, and volatilities are key steps in order to achieve successful reuse in the development of a product line. The adequate selection of the rights products can be regarded as a key factor of success for introducing the product line approach in a company. In this tutorial, we show how to identify and analyze emergent features in the future, and distinguish unstable from stable features. The aim is to prepare the product line for likely future adaptation needs by planning for changes beforehand.

This tutorial covers product line scoping and product line evolution planning, by discussing the following topics:

- The importance of product line scoping, analysis, and modeling as a key factor for successful product line engineering.
- Key principles of scoping (e.g. common and variable features, domains, and products).
- The PuLSE-Eco approach [1][2] and an overview of the approaches mentioned in [3].
- Key concepts for predicting future adaptation needs, and consequent likely changes in product lines.
- The PLEvo-Scoping approach for integrated and early evolution planning in product line engineering [4].

These topics address technological, organizational, and business aspects of the introduction of a product line, giving practitioners a solid basis to start with. The intended

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 515–516, 2010.

audience is practitioners that want to learn how to scope product lines and manage their evolution successfully, as well as researchers that want to know about an integrated approach for product line scoping and planning for future evolution.

2 Plan

The tutorial consists of three parts:

1. Introduction
   Overview of product line scoping, analysis, and modeling in an architecture centric product line approach. Importance of a thoroughly planned product line as a key factor for successful product line engineering and evolution. Key principles of product line requirements engineering.
2. Scoping
   Overview of existing scoping approaches, based on [3]. Introduction to the PuLSE-Eco approach for scoping, which includes explanation about the activities and key principles of scoping. Explanation on how to build up key artifacts (e.g. a product–feature matrix).
3. Product Line Evolution
   Overview of a model of software evolution, which defines key concepts for systematic reasoning on product line requirements volatility. Introduction to PLEvo-Scoping, a method based on such concepts, encompassing its activities and a complete example of its usage. Integration with existing scoping approaches.

The tutorial is based on our experience with product line engineering in many industrial projects. It crystallizes the essence of the experience gained in those industrial projects, and combines it with our latest research in the area of evolution in product line engineering.

Acknowledgements

This work was partially supported by the Fraunhofer Innovation Cluster for Digital Commercial Vehicle Technology (DNT/CVT Cluster).

References

[1] John, I., Knodel, J., Lehner, T., et al.: A Practical Guide to Product Line Scoping. In: Proc. SPLC 2006, Baltimore, pp. 3–12 (2006)
[2] Schmid, K.: Planning Software Reuse – A Disciplined Scoping Approach for Software Product Lines. Fraunhofer IRB Verlag, Stuttgart (2003)
[3] John, I., Eisenbarth, I.M.: A Decade of Scoping- a Survey. In: Proceedings of SPLC 2009, pp. 31–40 (2009)
[4] Villela, K., John, I., Dörr, J.: Evaluation of a Method for Proactively Managing the Evolving Scope of a Software Product Line. In: Wieringa, R., Persson, A. (eds.) REFSQ 2010. LNCS, vol. 6182, pp. 113–127. Springer, Heidelberg (2010)
[5] Bayer, J., Flege, O., Knauber, P., et al.: PuLSE: A Methodology to Develop Software Product Lines. In: Proc. SSR 1999, Los Angeles, pp. 122–131 (1999)

Leveraging Model Driven Engineering in Software Product Line Architectures

Bruce Trask and Angel Roman

MDE Systems Inc.

Abstract. Model Driven Engineering (MDE) is a promising recent innovation in the software industry that has proven to work synergistically with Software Product Line Architectures (SPLAs). It can provide the tools necessary to fully harness the power of Software Product Lines. The major players in the software industry including commercial companies such as IBM, Microsoft, standards bodies including the Object Management Group and leading Universities such as the ISIS group at Vanderbilt University are embracing this MDE/PLA combination fully. IBM is spearheading the Eclipse Foundation including its MDE tools like EMF, GEF and GMF. Microsoft has launched there Software Factories foray into the MDE space. Top software groups such as the ISIS group at Vanderbilt are using these MDE techniques in combination with Software Produce Line Architectures for very complex systems. The Object Management Group is working on standardizing the various facets of MDE. All of these groups are capitalizing on the perfect storm of critical innovations today that allow such an approach to finally be viable. To further emphasize the timeliness of this technology is the complexity ceiling the software industry find itself facing wherein the platform technologies have increased far in advance of the language tools necessary to deal with them.

As more and more software products are or are evolving into families of systems, it is vital to formally capture the commonalities and variabilities, the abstractions and the refinements, the frameworks and the framework extension points and completion code associated with a particular family. Model Driven Engineering has shown to be a very promising approach to capturing these aspects of software systems and families of systems which thereafter can be integrated systematically into Software Product Lines and Product Line Architectures..

The process of Developing Software Product Line Architectures can be a complex task. However, the use of Model Driven Engineering (MDE) techniques can facilitate the development of SPLAs by introducing Domain Specific Languages, Graphical Editors, and Generators. Together these are considered the sacred triad of MDE. Key to understanding MDE and how it fits into SPLAs is to know exactly what each part of the trinity means, how it relates to the other parts, and what the various implementations are for each. This tutorial will demonstrate the use of the Eclipse Modeling Framework (EMF) and Eclipse's Graphical Editor Framework (GEF) to create an actual MDE solution as applied to a sample SPLA. When building Graphical Modeling Languages using GEF and EMF one may find themselves worrying about specific implementation details related to EMF or GEF. To address

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 517–518, 2010.

this issue, the Eclipse community has created a generative bridge between EMF and GEF called The Graphical Modeling Framework (GMF). During this tutorial we will also illustrate how to model the visual artifacts of our Domain Model and generate a Domain Specific Graphical Editor using GMF.

This tutorial continues to be updated each year to include recent and critical innovations in MDE and SPL. This year will include information on key Model Transformation and Software Product Line migration technologies as well as various Model Constraint technologies.

The goal of this tutorial is to educate attendees on what MDE technologies are, how exactly they relate synergistically to Product Line Architectures, and how to actually apply them using an existing Eclipse implementation. The benefits of the technology are so far reaching that we feel the intended audience spans technical managers, developers and CTOs. In general the target audience includes researchers and practitioners who are working on problems related to the design and implementation of SPLAs and would like to understand the benefits of applying MDE techniques towards SPLAs and leverage Eclipse as a framework to develop MDE solutions. The first half will be less technical than the second half where we cover the details of PLA and MDE in action in complete detail showing patterns and code.

Introduction to Software Product Lines Adoption

Linda M. Northrop and Lawrence G. Jones

Software Engineering Institute,
Carnegie Mellon University,
Pittsburgh, PA 15213
{lmn,lgj}@sei.cmu.edu

Abstract. This tutorial describes a phased, pattern-based approach to software product line adoption. It reviews the challenges of product line adoption; introduces a roadmap for phased, product line adoption; describes adoption planning artifacts; and establishes linkages with other improvement efforts.

1 Introduction

The tremendous benefits of taking a software product line approach are well documented. Organizations have achieved significant reductions in cost and time to market and, at the same time, increased the quality of families of their software systems. However, to date, there are considerable barriers to organizational adoption of product line practices. Phased adoption is attractive as a risk reduction and fiscally viable proposition.

2 Overview

This tutorial describes a phased, pattern-based approach to software product line adoption. This tutorial will acquaint participants with product line adoption barriers and two ways to overcome them:

1. a phased, pattern-based adoption approach
2. explicit linkage with other improvement efforts

The objectives of the tutorial are to acquaint participants with

- issues surrounding software product line adoption
- a phased, pattern-based adoption approach
- adoption planning artifacts
- explicit linkage of software product line adoption with other improvement efforts

The tutorial begins with a discussion of software product line adoption issues, including benefits, barriers, risks and the technical and organizational factors that influence adoption. We then present the Adoption Factory pattern [1]. The Adoption

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 519–520, 2010.

Factory pattern provides a roadmap for phased, product line adoption. The tutorial covers the Adoption Factory in detail, including focus areas, phases, subpatterns, related practice areas, outputs, and roles. Examples of product line adoption plans following the pattern are used to illustrate its utility [2]. The tutorial also describes strategies for creating synergy within an organization between product line adoption and ongoing CMMI or other improvement initiatives [3,4,5].

Participants should have experience in designing and developing software-intensive systems, some familiarity with modern software engineering concepts and management practices, and be familiar with product line concepts. The tutorial is aimed at those in an organization who are in a position to influence the decision to adopt a product line approach, and those in a position to carry out that decision. This includes technical managers at all levels, as well as those on the software development staff. Anyone who can act as a technology change agent will benefit from this tutorial.

References

1. Northrop, L.: Software Product Line Adoption Roadmap (CMU/SEI-2004-TR-022), Software Engineering Institute, Carnegie Mellon University (2004), http://www.sei.cmu.edu/publications/documents/04.reports/04tr022.html
2. Clements, P., Jones, L., McGregor, J., Northrop, L.: Getting From There to Here: A Roadmap for Software Product Line Adoption. Communications of the ACM (49, 12) (December 2006)
3. Jones, L., Northrop, L.: Product Line Adoption in a CMMI Environment (CMU/SEI-2005-TN-028), Software Engineering Institute, Carnegie Mellon University (2005), http://www.sei.cmu.edu/publications/documents/05.reports/05tn028.html
4. Jones, L.: Software Process Improvement and Product Line Practice : Building on Your Process Improvement Infrastructure (CMU/SEU-2004-TN-044), Software Engineering Institute, Carnegie Mellon University (2004), http://www.sei.cmu.edu/publications/documents/04.reports/04tn044.html
5. Jones, L., Soule, A.: Software Process Improvement and Product Line Practice: CMMI and the Framework for Software Product Line Practice (CMU/SEI-2002-TN-012), Software Engineering Institute, Carnegie Mellon University (2002), http://www.sei.cmu.edu/publications/documents/02.reports/02tn012.html

Introduction to Software Product Lines

Linda M. Northrop

Software Engineering Institute,
Carnegie Mellon University,
Pittsburgh, PA 15213
lmn@sei.cmu.edu

1 Introduction

Software product lines have emerged as a new software development paradigm of great importance. A software product line is a set of software intensive systems sharing a common, managed set of features, that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. Organizations developing a portfolio of products as a software product line are experiencing order-of-magnitude improvements in cost, time to market, staff productivity, and quality of the deployed products.

2 Overview

This tutorial introduces the essential activities and underlying practice areas of software product line development. It is aimed at those in an organization who are in a position to influence the decision to adopt a product line approach, and those in a position to carry out that decision. Anyone who can act as a technology change agent will benefit from this tutorial.

The tutorial reviews the basic concepts of software product lines, discusses the costs and benefits of product line adoption, introduces the SEI's Framework for Software Product Line Practice[1] [1], and describes approaches to applying the practices of the framework.

This is a half-day introduction to software product lines. It provides a quick overview rather than a detailed exposition. The tutorial is based on the book Software Product Lines: Practices and Patterns [2] and includes material on the following topics.

Software product line concepts

What software product lines are and aren't; basic terminology; strategic reuse; success stories

Costs and benefits

Economics of software product lines; individual and organizational benefits

[1] SEI Framework for Software Product Line Practice is a service mark of Carnegie Mellon University.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 521–522, 2010.

The three essential activities
Core asset development; product development; organizational and technical management

Product line practice areas
Practice area categories; the SEI's Framework for Software Product Line Practice [1]

Making it happen
Applying the practice areas; gauging an organization's ability to succeed with software product lines

Wrap-up
Current research and remaining challenges

References

1. Northrop, L., Clements, P.: A Framework for Software Product Line Practice (2010), http://www.sei.cmu.edu/productlines/framework.html
2. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston (2002)

4th International Workshop on Dynamic Software Product Lines (DSPL 2010)

Svein Hallsteinsen[1], Mike Hinchey[2], Sooyong Park[3], and Klaus Schmid[4]

[1] SINTEF ICT, Norway
[2] Lero – The Irish Software Engineering Research Centre, Limerick, Ireland
[3] University of Sogang, South Korea
[4] University of Hildesheim, Germany

As software systems increase in complexity and size their capabilities in adapting to a wide range of situations becomes increasingly important. Product line techniques can be used to successfully such flexibility even at after development time, like at initialization or even runtime. Over the last few years, this branch of software systems has established itself successfully as dynamic software product lines (DSPL).

In a DSPL various software variants are managed, variation points are bound flexibly, but all this is done at runtime, in the most extreme cases this can mean that the whole product derivation is fully automatic and that reconfiguration can happen repeatedly at runtime. DSPL is strongly related to current research topics like self-managing systems, autonomous, ubiquitous systems, etc. However, it adds product line engineering techniques, methods and processes to the mix as a conceptual basis.

The objective of this workshop is to provide a forum for the exchange of ideas, to establish cooperations, and to bring industry and research together.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, p. 523, 2010.

1st International Workshop on Product-Line Engineering for Enterprise Resource Planning (ERP) Systems (PLEERPS 2010)

Haitham S. Hamza[1] and Jabier Martinez[2]

[1] Cairo University and Software Engineering Competence Center (SECC), Egypt
[2] European Software Institute (ESI-Tecnalia), Spain

Enterprise resource planning (ERP) is an industry term commonly used to describe a business management system that integrates activities across functional departments including planning, manufacturing, purchasing of parts, controlling and maintaining inventory, tracking orders, etc. Return on Investment (ROI) and Return on Values (ROV) for developing ERP systems largely depend on the capability of evolving, maintaining, and customizing/configuring ERP systems to respond to new business needs and emerging market segments.

Software Product Line (SPL) fits naturally into the ERP business. ERP systems can benefit greatly from the concepts of commonalities and variabilities to enhance evolutionability and maintainability. Moreover, product-line concepts can substantially reduce current tedious configuration procedures that are not only resource-intensive, but also error-prone.

The central theme of this workshop is to bring together researchers and practitioners in SPL and ERP industry to report on their experience and investigate current and future practical challenges in order to adopt product line architectures (PLAs) for developing ERP systems. A broader objective of this workshop is to investigate and to document practical experiences in adopting PLAs in different domains. We believe that, sharing the challenges and barriers in introducing PLAs to a concrete domain can be useful for practitioners across various domains.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, p. 524, 2010.

2nd International Workshop on Model-Driven Approaches in Software Product Line Engineering (MAPLE 2010)

Deepak Dhungana[1], Iris Groher[2], Rick Rabiser[2], and Steffen Thiel[3]

[1] Lero, University of Limerick, Limerick, Ireland
[2] Johannes Kepler University, Linz, Austria
[3] Furtwangen University of Applied Sciences, Furtwangen, Germany

Software product line engineering (SPLE) promises order-of-magnitude improvements in time-to-market, cost, productivity, quality, and other business drivers. Many of these expected benefits are based on the assumption that the additional investment for setting up a software product line pays off during application engineering. However, to fully exploit this we need to optimize application engineering processes and systematically and efficiently handle reusable artefacts. The goal of this workshop is to explore and explicate the current status within the field of model-driven approaches in SPLE. The workshop has the following aims: (i) Demonstrate how model-driven concepts and techniques can be applied in SPLE; (ii) Show how model-driven approaches have already successfully been applied; (iii) Explore how models with a sound semantic foundation can facilitate automated and interactive SPLE.

Workshop topics include but are not limited to: modelling of software product lines; product derivation and configuration; aspect-oriented approaches; evolution and change; validation and verification of models; scalability of modelling approaches; modelling in the context of software ecosystems; integrating modelling approaches; industrial experiences; applied formal methods for models; dynamic architectures and variability; software and service configuration approaches.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, p. 525, 2010.

1st International Workshop on Formal Methods in Software Product Line Engineering (FMSPLE 2010)

Ina Schaefer[1], Martin Becker[2], Ralf Carbon[2], and Sven Apel[3]

[1] Chalmers University of Technology, Gothenburg, Sweden
[2] Fraunhofer IESE, Kaiserslautern, Germany
[3] University of Passau, Germany

Software product line engineering (SPLE) aims at developing a family of systems by reuse in order to reduce time to market and to increase product quality. The correctness of the reusable development artifacts as well as the correctness of the developed products is of crucial interest for many safety-critical or business-critical applications. Formal methods have been successfully applied in single system engineering over the last years in order to rigorously establish critical system requirements. However, in SPLE, formal methods are not broadly applied yet, despite their potential to improve product quality. One of the reasons is that existing formal approaches from single system engineering do not consider variability, an essential aspect of product lines.

The goal of the workshop "Formal Methods in Software Product Line Engineering (FMSPLE)" is to bring together researchers and practitioners from the SPLE community with researchers and practitioners working in the area of formal methods. The workshop aims at reviewing the state of the art in which formal methods are currently applied in SPLE in order to initiate discussions about a research agenda for the extension of existing formal approaches and the development of new formal techniques dealing with the particular needs of SPLE.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, p. 526, 2010.

3rd International Workshop on Visualisation in Software Product Line Engineering (VISPLE 2010)

Steffen Thiel[1], Rick Rabiser[2], Deepak Dhungana[3], and Ciaran Cawley[3]

[1] Furtwangen University of Applied Sciences, Furtwangen, Germany
[2] Johannes Kepler University, Linz, Austria
[3] Lero, University of Limerick, Limerick, Ireland

To leverage the explicit and extensive reuse of shared software artefacts, many companies use a product line approach to build different variants of their products for use within a variety of systems. Product lines can be large and could easily incorporate thousands of elements together with diverse relationships among them. This makes product line management and systematic product derivation extremely difficult.

This workshop aims at elaborating on the idea of using information and software visualisation techniques to achieve the economies of scale required to support variability management and product derivation in industrial product lines. Visualisation techniques have been proven to support improvement in both the human understanding and effective use of computer software. The workshop aims to explore the potential of visual representations combined with the use of human interaction techniques when applied in a software product line context.

Topics of interest focus on visualisation and interaction techniques and tools for managing large and complex software product lines as well as related industrial experience; variability representation, i.e., techniques and tools for coping with large numbers of variation points; visualisation of large data sets, i.e., general software visualisation techniques and tools for managing large data sets and their applicability in a software product line context; representation of features and software architecture; traceability visualisation, i.e., techniques and tools to represent trace links in and among problem and solution space; visualisation and interaction techniques and tools to support product derivation and product line maintenance and evolution.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, p. 527, 2010.

4th Workshop on Assessment of Contemporary Modularization Techniques (ACOM 2010)

Alessandro Garcia[1], Phil Greenwood[2], Yuanfang Cai[3], Jeff Gray[4], and Francisco Dantas[1]

[1] PUC-Rio, Brazil
[2] Lancaster University, UK
[3] Drexel University, USA
[4] University of Alabama, USA

Various modularization techniques are prominent candidates for taming the complexity of software product lines (SPLs), such as aspect-oriented software development (AOSD) and feature-oriented software development (FOSD). The ACoM workshop aims to bring together researchers and industrial practitioners with different backgrounds to: (a) understand the role and impact of contemporary modularization techniques on the design and implementation of SPLs; (b) explore new, and potentially more effective, assessment methods to guide the application of modularization techniques in SPL development, and (c) discuss the potential of using modularity assessment results to improve SPL development outcomes, to modularization techniques, and to foster the development of new techniques. This is the fourth edition of the ACoM workshop, an itinerant event previously held at ICSE and OOPSLA.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, p. 528, 2010.

2nd Workshop on Scalable Modeling Techniques for Software Product Lines (SCALE 2010)

M. Ali Baba[1], Sholom Cohen[2], Kyo C. Kang[3], Tomoji Kishi[4], Frank van der Linden[5], Natsuko Noda[6], and Klaus Pohl[7]

[1] IT University of Copenhagen, Denmark
[2] SEI, USA
[3] POSTEC, Korea
[4] Waseda University, Japan
[5] Philips, The Netherlands
[6] NEC Corporation, Japan
[7] University of Duisburg-Essen, Germany

Modeling techniques play essential roles in software product line development (PLD), and various modeling techniques have been proposed so far. However, some of these techniques are not actually usable in the industries, due to the lack of scalability. Although modeling techniques are essentially for reducing scale and complexity, further development of techniques are indispensable to manage the scale and complexity we are confronting today. Especially, in PLD, the problem becomes more serious, because we have to model target domains, requirements, architectures, designs along with complicated variabilities and configurations.

The objective of this workshop is to bring together both researchers and practitioners to discuss the strengths and limitations of the current modeling techniques for supporting large-scale PLD. At the last SCALE workshop, participants brought various scalability problems and we categorized important issues in this field. These are the basis of our next discussion. We also have to consider the recent business and technical situations that demand scalable modeling such as distributed development, offshore development, core asset evolution and cloud computing. Based on above, we plan to make further analysis of scalability problems and examine modeling techniques for each scalability issue.

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, p. 529, 2010.

Author Index