Benavides, David (Ed.); Metzger, Andreas (Ed.); Eisenecker, Ulrich (Ed.)

Research Report

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

ICB-Research Report, No. 29

Provided in Cooperation with:
University Duisburg-Essen, Institute for Computer Science and Business Information Systems (ICB)

Suggested Citation: Benavides, David (Ed.); Metzger, Andreas (Ed.); Eisenecker, Ulrich (Ed.) (2009) : Third International Workshop on Variability Modelling of Software-intensive Systems. Proceedings, ICB-Research Report, No. 29, Universität Duisburg-Essen, Institut für Informatik und Wirtschaftsinformatik (ICB), Essen

This Version is available at:
https://hdl.handle.net/10419/58158

ICB
Institut für Informatik und Wirtschaftsinformatik

David Benavides
Andreas Metzger
Ulrich Eisenecker (Eds.)

Third International Workshop on Variability Modelling of Software-intensive Systems

ICB-RESEARCH REPORT 29

Proceedings

UNIVERSITÄT DUISBURG ESSEN

ICB-Research Report No. 29

January 2009

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

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

Proceedings

Edited By:

David Benavides

University of Seville, Spain

benavides@us.es

Andreas Metzger

University of Duisburg-Essen, Germany

andreas.metzger@sse.uni-due.de

Ulrich Eisenecker

University of Leipzig, Germany

eisenecker@wifa.uni-leipzig.de

Contact:

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

Tel.: 0201-183-4041
Fax: 0201-183-4011
Email: icb@uni-duisburg-essen.de

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

Abstract

This ICB Research Report constitutes the proceedings of the Third International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'09), which was held from January 28–30, 2009 at the University of Sevilla, Spain.

Table of Contents

1 Message from the Organizers

Welcome to the third International Workshop on Variability Modelling of Software-intensive Systems: VaMoS'09!

Previous VaMoS workshops have been held in

- Limerick (2007) and
- Essen (2008).

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

Continuing the successful format of the two previous VaMoS workshops, VaMoS 2009 will be a highly interactive event. Each session will be organized in such a way that discussions among the workshop participants will be stimulated. We hope that VaMoS will trigger work on new challenges in variability modelling and thus will help to shape the future of variability modelling research.

VaMoS'09 has attracted 32 submissions of authors from 15 countries. Each submission was reviewed by at least three members of the programme committee. Based on the reviews, 18 submissions have been accepted as full papers and 4 submissions have been accepted as short papers documenting tool demonstrations.

We like to extend our gratitude to all the people who spent time and energy to make VaMoS a success. VaMoS'09 would not have been possible without their efforts and expertise. We are thankful to Don Batory who accepted our invitation to give a keynote talk on "Dimensions of Variability in Software Product Lines". We cordially thank all the members of the VaMoS programme committee for devoting their time to reviewing the submitted papers. We are grateful to the people who helped preparing and organizing the event, especially Sergio Segura, Pablo Trinidad, Adela del Río, Octavio Martín and Manuel Resinas. Finally, we thank the sponsors of VaMoS: The University of Sevilla, The University of Leipzig and the University of Duisburg-Essen.

Enjoy VaMoS 2009 and a beautiful Sevilla!

The VaMoS organizers

David Benavides

Andreas Metzger

Ulrich Eisenecker

2 Organization

Organizing Committee

David Benavides, University of Seville, Spain
Andreas Metzger, University of Duisburg-Essen, Germany
Ulrich Eisenecker, University of Leipzig, Germany

Steering Committee

Klaus Pohl, University of Duisburg-Essen, Germany
Patrick Heymans, University of Namur, Belgium
Kyo-Chul Kang, Pohang University of Science and Technology, Korea

Programme Committee

Don Batory, University of Texas, USA
Jürgen Börstler, Umeå University, Sweden
Manfred Broy, TU Munich, Germany
Pascal Costanza, Free University of Brussels, Belgium
Oscar Díaz, Universidad del País Vasco, Spain
Xavier Franch, Universidad Politécnica de Catalunya, Spain
Stefania Gnesi, ISTI-CNR, Italy
Paul Gruenbacher, Johannes Kepler Universitat Linz, Austria
Øystein Haugen, University of Oslo & SINTEF, Norway
Tomoji Kishi, Japan Advanced Institute of Science and Technology
Roberto Lopez-Herrejon, Bournemouth University, UK
Tomi Männistö, Helsinki University of Technology, Finland
Kim Mens, Université catholique de Louvain, Belgium
Dirk Muthig, Fraunhofer IESE, Germany
Linda Northrop, SEI, USA
Vicente Pelechano, Universidad Politécnica de Valencia, Spain
Antonio Ruiz-Cortés, Universidad de Sevilla, Spain
Camille Salinesi, University of Paris 1-Sorbonne, France
Klaus Schmid, University of Hildesheim, Germany
Doug Schmidt, Vanderbilt University, USA
Vijay Sugumaran, Oakland University, USA
Steffen Thiel, Lero, Limerick, Ireland
Frank van der Linden, Philips, The Netherlands
Jilles van Gurp, Nokia Research, Finland
Matthias Weber, Carmeq GmbH, Germany
Liping Zhao, University of Manchester, UK

3 Workshop Format

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

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

1) Presenter

A presenter obviously presents his paper but additionally will be asked to take on the role of discussant for the other paper in his session. It is highly desired that – as a presenter – you attend the complete event and take an active part in the discussion of the other papers. Prepare your presentation and bear in mind the available time, which is **15 min** for the paper presentation.

2) Discussant

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

3) Session Chair

A session chair's tasks are as follows:

Before the session starts:

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

For each paper presentation:

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

After the last presentation:

- Lead through and moderate the general discussion.
- Finally, close the session when the allotted time has elapsed.

4 Technical Programme

Keynote

Research Papers (Full Papers)

Tool Demonstrations (Short Papers)

Dimensions of Variability in Software Product Lines

Don Batory
Department of Computer Science
University of Texas at Austin
Austin, Texas 78712
batory@cs.utexas.edu

Abstract

Transformation-based program synthesis is a hallmark of automated program development. Some time ago, we discovered that the design of a program could be expressed as a matrix of transformations, where both rows and columns represented features. The technique was called Origami, as the matrix was folded in precise ways (thereby composing transformations) until a scalar was produced. This scalar defined an expression (a composition of transformations) that, when evaluated, synthesized the program. Origami generalized to n-dimensional matrices, where each axis defined a dimension of variability. But we never quite understood why Origami worked.

Our research seeks principles of automated construction that can be appreciated by practitioners and that are expressed in terms of simple mathematics. This talk explains Origami by an interesting integration of diverse topics: data cubes (database technology), basic ideas from tensors and categories (mathematics), extensibility problem (programming languages), and feature interactions (software design).

Comparitive Study of Variability Management in Software Product Lines and Runtime Adaptable Systems

Vander Alves, Daniel Schneider, Martin Becker
Fraunhofer IESE
Fraunhofer Platz 1, 67663 Kaiserslautern, Germany
<*first name*>.<*last name*>@iese.fraunhofer.de

Nelly Bencomo, Paul Grace
Computing department, InfoLab21, Lancaster University,
Lancaster, LA1 4WA, United Kingdom
{nelly, gracep}@comp.lancs.ac.uk

Abstract

Software Product Lines (SPL) and Runtime Adaptation (RTA) have traditionally been distinct research areas addressing different problems and with different communities. Despite the differences, there are also underlying commonalities with synergies that are worth investigating in both domains, potentially leading to more systematic variability support in both domains. Accordingly, this paper analyses commonality and differences of variability management between SPL and RTA and presents an initial discussion on the feasibility of integrating variability management in both areas.

1. Introduction

Software Product Line (SPL) [15] and Runtime Adaptation (RTA) [35] have traditionally been distinct research areas addressing different problems and with different communities (e.g., SPLC and ICSR in the former area and Middleware in the latter). SPL deals with strategic reuse of software artifacts in a specific domain so that shorter time-to-market, lower costs, and higher quality are achieved. In contrast to that, RTA aims for optimized service provisioning, guaranteed properties, and failure compensation in dynamic environments. To this end, RTA deals mostly with dynamic flexibility so that structure and behaviour is changed in order to dynamically adapt to changing conditions at runtime.

Despite the differences, there are also underlying commonalities with synergies that are worth investigating across both domains. For instance, in terms of commonalities, both areas deal with adaptation of software artifacts: by employing some variability mechanism applied at a specific binding time, a given variant is instantiated for a particular context. Accordingly, the research community has recently begun to explore the synergies between these two research areas.

On the one hand, motivated by the need of producing software capable of adapting to fluctuations in user needs and evolving resource constraints [27], SPL researchers have started to investigate how to move the binding time of variability towards runtime [4, 11, 33], also noticeable in the research community with even specific venues, such as the Dynamic Software Product Line (DSPL) workshop at SPLC, currently in its second edition. On the other hand, motivated by the need of more consolidated methods to systematically address runtime variability, RTA researchers have started to investigate leveraging SPL techniques [24, 14, 10].

Nevertheless, in either case, a refined and systematic comparison between these two areas is still missing. Such comparison could help to explore their synergy with cross-fertilization that could lead to more systematic variability support in both domains.

In this context, this paper presents two key contributions:

- it analyses commonality and differences of variability management between SPL and RTA. We define variability management as the handling of variant and common artifacts during software lifecycle including development for and with reuse. We choose variability management because we see it as the common denominator for exploring synergies between SPL and RTA;
- it presents an initial discussion on the feasibility of integrating variability management in SPL and RTA.

The remainder of this paper is structured as follows. Sections 2 and 3 briefly review conceptual models for variability management in SPL and RTA. Next, Section 4 presents comparison criteria and compares variability management between SPL and RTA approaches. Section 5 then discusses potential integration of SPL and RTA. Related work is considered in Section 6, and Section 7 offers concluding remarks.

2. Software Product Line Variability Management

There are different approaches for describing SPL variability, for instance Orthogonal Variability Model [38] and PuLSE's meta-model [7]. In this work, we comply with the latter, since it has been applied in research and industrial projects for years. In particular, Figure 1 depicts the conceptual model for a SPL, with a focus on variability management. The figure is based on a simplified version of the meta-model proposed by Muthig [36], highlighting some parts of the instantiation process and the actors involved.

A SPL comprises a set of products and a SPL infrastrucutre developed in a specific domain. The first are developed by the application engineer, whereas the latter are developed by the domain engineer and are reused in more than one product. The SPL infrastructure consists of SPL assets, which in turn comprise a decision model and SPL artifacts. A special kind of PLAsset is the PLArchitecture, which represents the SPL reference architecture. SPL artifacts are generic SPL artifacts, i.e., they embed variation points which have an associated binding time and can be described according to a given mechanism. A decision model represents variability in a SPL in terms of open decisions and possible resolutions. In a decision model instance, known as Product Model, all decisions are resolved, which is used to instantiate a specific product from SPL artifacts [6].

A product consists of product artifacts in a given context. A product artifact is an instance of SPL artifacts and comprises Variants, which in turn are instances of variation points after these have been resolved by the decision model when the application engineer configures this model into the product model.

Binding time refers to the time at which the decisions for a variation point are bound [32]. Examples of binding time are pre-compilation, compilation, linking, load, or runtime. Traditionally, SPL has been used mostly without the latter binding time. Therefore, in those cases, with instantiation of the product, variability is bound and a specific running application is obtained. Variation points and decision models then do not persist in the generated product.

Another artifact that does not persist in the generated product is context. We adopt the general definition of context proposed by Dey et al. [18]:"*Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves*". For example, context can be language or regulations. Although context is considered during domain analysis and is used by the decision model, it is not explicitly integrated into the instantiated product. Further, it is often not formally represented; rather, it is usually informally available to the domain and application engineers in non-computable form [17].

Since variation points, decision models, and context usually do not exist in the instantiated SPL products, switching from one product to another product is only possible at earlier binding times. The corresponding transition then is not from one variant to another, but from the design space to a variant and requires direct intervention of the application engineer.

3. Runtime Adaptation Variability Management

In this paper, an adaptable system is a system that can be changed at runtime to better meet users' needs, and an adaptive system is a systems that changes by itself at runtime. The discussion in this paper focuses more in adaptive systems. The domain model for runtime adaptation showing the concepts that are relevant to this paper is depicted in Figure 2. The domain model is based on previous work on developing Dynamically Adaptive Systems (DASs) in middleware research [10, 26]. We define a DAS as a software system with enabled runtime adaptation. Runtime adaptation takes place according to context changes during execution. Example of DASs we have developed at Lancaster University are the adaptive flood warning system deployed to monitor the River Ribble in Yorkshire, England [30, 29, 10]; and the service discovery application described in [16]. The figure serves as a conceptual model to help explain the description that follows.

A Reference Architecture addresses specific solution Domains, such as routing algorithms, networking technologies, and service discovery. The Reference Architecture is specified by the Domain Engineer. DAS will typically employ a number of System Variants, which are sets of Component Configurations. Different component configurations can result in different connection topologies (compositions) as well as in different "internal" behaviour (parameters). Dynamic adaptation is achieved via transitions between Component Configuration variations over time; for example, components being added, removed and replaced, as the DAS adapts based upon environmental context changes. In any case, every Component Configuration must conform to

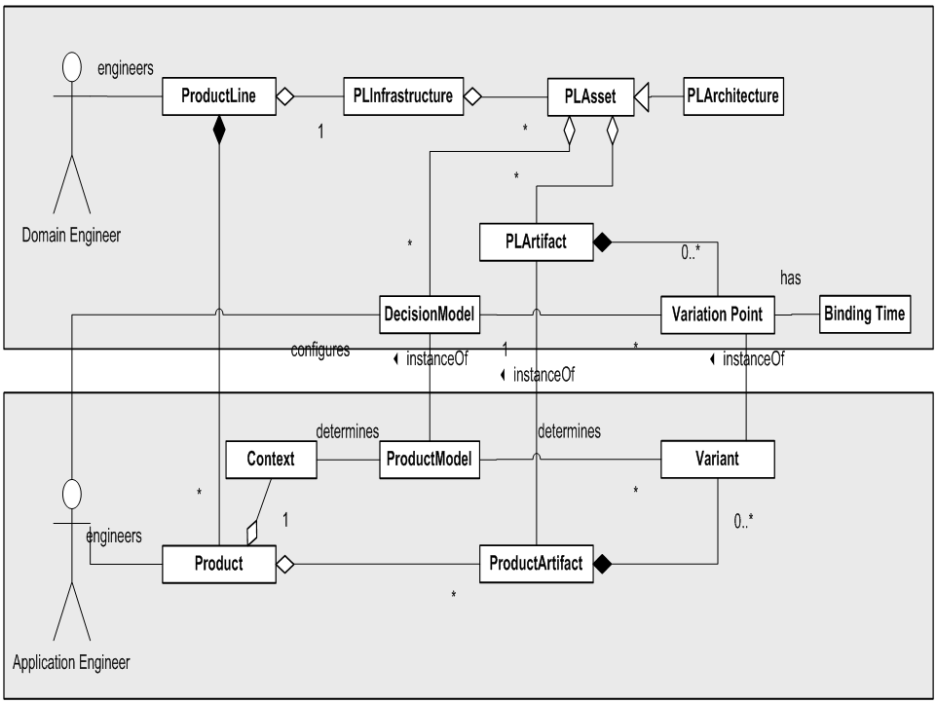


Figure 1. Variability Management in Software Product Lines.

the Reference Architecture, which describes the structural commonalities of a DAS that always hold.

In addition to models describing the configuration space at runtime, we require means to identify situations when to adapt and which configuration to choose. This is represented in Figure 2 by the transitions. A transition starts in a previous system variant and ends in a next variant. Transitions occur due to Triggers. Triggers are specified by the Application Engineer in terms of conditions of environment and context. We distinguish between two different types of approaches on how the best system variant can be determined [41]:

1. Rule-based approaches [10, 41, 43] usually have the "Event-Condition-Action" (ECA) form and hence distinctly specify when to adapt and which variant to choose. Such approaches are widely spread in the domains of sensor computing, embedded systems, and mobile computing. One reason is that such systems need to rely on light-weight approaches due to the inherent scarcity of resources (e.g., processing time, power supply) and must be deterministic. The rule sets are usually to be specified at design time. However, rules can also be added during execution [10].

2. Goal-based approaches [28, 34, 39] equip the system with goal evaluation functions in order to determine the best system variant under current circumstances. Neglecting available optimization strategies, the brute force approach would determine and evaluate all currently valid system variants and choose the variant that meets best the given goals. Thus, goal-based adaptation can be more flexible than rule-based adaptation and it is more likely that optimal configurations can be identified, albeit at a higher resource usage.

For example, reconfiguration policies can take the form of ECA rules. Actions are changes to component configurations while events in the environment are notified by a context manager. System Variants will change in response to changes of Environment Variants. Environment Variants represent properties of the environment that provide the context for the running system. Different values associated with these properties define the possible triggers of the transitions. System Variants represent different configurations permissible within the constraints of the DASs Reference Architecture. Environment Variants are specified by Domain Engineers. The variation points associated with the component configurations are specified using orthogonal variability models [38], which are not described in Figure 2 but addressed elsewhere [10]. For the DAS to operate in the context of any Environment Variant, it needs to be configured as the appropriate System Variant.

Dynamic reconfiguration can be performed via the dy-

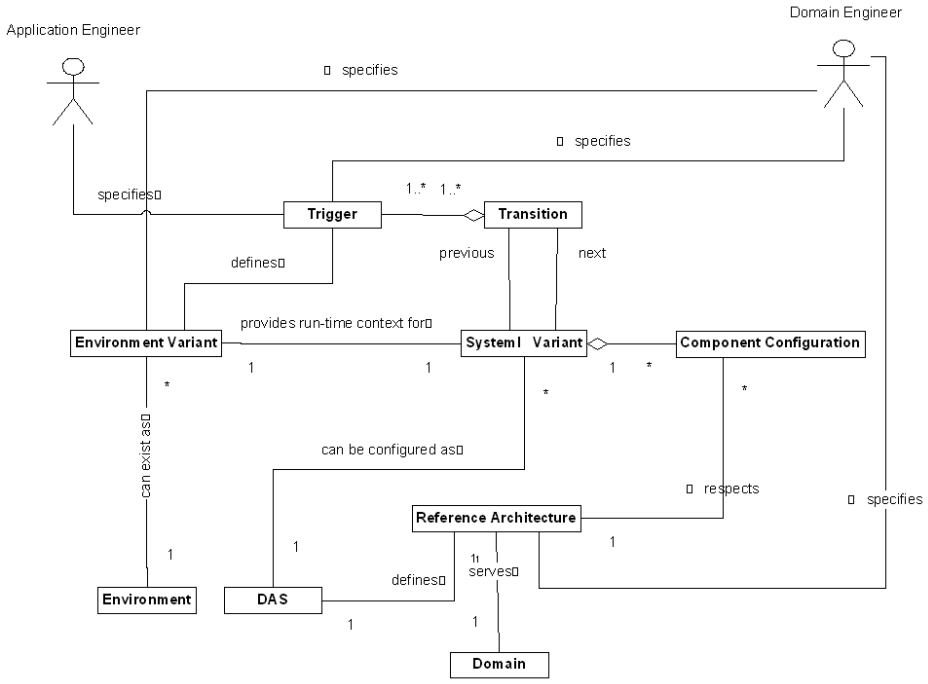


Figure 2. Variability Management in runtime adaptable system.

namic composition of components at runtime. However, the methodology is equally applicable to other runtime composition mechanisms, e.g., dynamic AOP.

In RTA, systems may need to adapt to the prevailing situation on their own. Therefore there needs to be runtime models defining when and how to adapt. Further, there needs to be a notion of context, as acceptable configurations strongly depend upon the prevailing situation. We consider the following aspects as typical drivers for runtime adaptation: changes in required functionality and Quality of Service (QoS), including dependability, changes in the availability of services and resources, and occurrence of failures.

4. Comparison of Variability Management between SPL and RTA

After briefly presenting conceptual models of variability for SPL and RTA in Sections 2 and 3, respectively, we now compare both domains. First, we present comparison criteria (Section 4.1) and then perform the comparison (Section 4.2). The result of the comparison is later discussed in Section 5 for potential synergies.

4.1 Comparison Criteria

With the goal of identifying synergy points in variability management between SPL and RTA, the comparison criteria we use are based on previous work by Becker et al. [8] and by McKinley et al. [35], both in the context of RTA, and on the taxonomy proposed by Svahnberg et al. [40] in the context of SPL. The criteria are as follows:

- **Goal**: addresses the goal(s) of variability. If there is no goal or driver for variability, the rest of the criteria are irrelevant. Goals are generally expressed as statements over a variability driver. Some examples of variability drivers are the following: functionality, quality (dependability), resources (e.g., CPU, memory, communication, display, time, energy), context (e.g., regulation, culture, language). A goal, for instance, is to improve dependability or optimize resource usage;

- **Binding time**: time when the variability decisions are resolved, e.g., pre-compilation, compilation, linking, load, or runtime;

- **Mechanism**: a software development technique that is used to implement a variation point, e.g., parametric polymorphism, subtype polymorphism, design patterns, aspects, conditional compilation, reflection,

selection of one among alternative implementations, frames;

- **Who**: defines who resolves the variability decisions. It can be the domain engineer, the application engineer, the user, or the application itself;

- **Variability Models**: define the models that have variation points or that control adaptation, e.g., decision model, reference architecture, PLArtifact, reconfiguration policies.

4.2 Comparison Results

According to the comparison criteria presented in the previous section, Table 1 compares variability management in RTA and SPL.

In terms of **goals**, although the fulfilment of functional and non-functional requirements is common in both SPL and RTA, SPL has focused more on providing fulfilment of functional requirements than RTA, and RTA has focused more on improving QoS than SPL. However, recently there has been a trend for SPL research to address QoS more closely [9, 22, 21, 24], which still remains an open issue and thus an explicit submission topic in key venues, e.g., SPLC09 [1]. Conversely, in RTA plugin architectures or component frameworks have enabled inclusion of new functionality into systems, e.g., web browser plugins.

Binding time in SPL has traditionally been at precompile, compile, and link time, whereas in RTA variability has been achieved at load time when the system (or component configuration) is first deployed and loaded into memory, and more commonly at runtime after the system has begun executing. The earlier binding in SPL usually allows for some static optimization and is thus usually more suitable for resource constrained systems, whereas the late binding time in RTA favours flexibility instead. Nevertheless, as mentioned previously, binding time in SPL has started to shift also to runtime in the context of DSPLs. Additionally, SPLs have also been built with flexible binding times, i.e., variation points that can have different binding times and binding times selected based on domain-specific context [12].

SPL **mechanisms** include diverse mechanisms such as conditional compilation, polymorphism, Aspect-Oriented Programming (AOP), Frames, parameterization, for example [3], and can be classified according to introduction times, open for adding variant, collection of variants, binding times, and functionality for binding [40]. On the other hand, at the core of all approaches to RTA adaptation is a level of indirection for intercepting and redirecting interactions among program entities [35]. Accordingly, key technologies are computational reflection, AOP, and component-based design. Examples of corresponding techniques are Meta-Object Protocols (MOP) [31], dynamic aspect weaving, wrappers, proxies, and architectural patterns (such as the Decentralized Control reconfiguration pattern [25]). RTA mechanisms can be described according to the taxonomy by McKinley et al. [35], which highlights how, when, and where to compose adaptations. Key technologies and techniques for RTA variability can also be used for SPL variability, but in cases where runtime binding time is not required this leads to suboptimal resource usage, since variation points persist unnecessarily. Nevertheless, not all SPL variability mechanism can be used for addressing variability in RTA, e.g., conditional compilation. Additionally, SPL mechanisms allow transitions from PLArtifact to Product Artifact at early binding time, whereas in RTA transitions occur from component configuration to another component configuration at runtime.

In SPL it is the application engineer **who** is responsible for resolving and implementing variability decisions. This includes the instantiation of corresponding PLArtifacts (with aid of the decision model), the development of product-specific artifacts and their integration. The responsible entity in RTA depends on the actual binding time. It is an expert/user at load time and the system itself at runtime. Consequently, the system requires means to perform the role of the application engineer during runtime (and partially so at load time), when, due to context changes, reconfiguration is necessary so that a new variant is generated. As mentioned in Section 3, the application engineer in RTA only specifies the triggers, but does not actually perform the adaptation. Instead, triggers themselves play this role.

In terms of **variability models**, SPL involves using mechanisms to adapt PLArtifacts according to the decision model, whereas RTA mechanisms adapt System Variants according to the configuration models and the reconfiguration policies. These variants are then an instance of the Reference Architecture. RTA variability models are inherently available at runtime, thus requiring an explicit and computationally tangible representation of all such artifacts, whereas variability SPL artifacts in general do not have a runtime representation and are often expressed informally.

5. Analysis and Potential Synergy

Based on the comparison from the previous section, we can now highlight some commonalities between SPL and RTA variability management. This is essential to foster potential synergy and cross-fertilization of best practices in both research areas, which is feasible given the recent interest in DSPLs [27].

As mentioned in Section 4.2, the distinction between variability management **goals** of both areas has become blurred. Since SPL now addresses QoS more commonly,

Criteria	SPL	RTA
Goal	Focus on functional requirements	Focus on improving QoS while maintaining functional requirements
Binding time	Mostly Pre-process/Compile/Linking	Load time/Runtime
Mechanism	e.g., conditional compilation, polymorphism, AOP, Frames, parameterization	e.g., MOP, dynamic aspect weaving, wrappers, proxies
Who	Application Engineer	Expert/User, Application itself
Variability Models	Decision Model, PLArtifact	Reference archtecture, System Variant, Variability rules

Table 1. Comparison of Variability Management between SPL and RTA.

it could benefit from well-established techniques for guaranteeing QoS at runtime that have been used in RTA. For example, Adapt [23] is an open reflective system that inspects the current QoS and then uses MOPs to alter the behaviour of the system through component reconfiguration if the required level of service is not maintained. Additionally, hybrid feature models, incorporating both functionality and QoS have also been proposed, e.g., by Benavides [9, 22, 21]. Conversely, RTA can use models for describing variability, such as enhanced versions of feature models [33] suitable for dynamic reconfiguration. Nevertheless, in this latter, there is still the challenge of addressing QoS issues [33].

Runtime binding time is on the focus of current research in SPL [33, 42, 4] and could leverage corresponding mechanisms in RTA. Wolfinger et al. [42] demonstrate the benefits of supporting runtime variability with a plug-in platform for enterprise software. Automatic runtime adaptation and reconfiguration are achieved by using the knowledge documented in variability models. Wolfinger et al. use the runtime reconfiguration and adaptation mechanism based on their own plug-in platform, which is implemented on the .NET platform.

In addition to the focus on runtime binding time in SPL, the *transition* itself towards runtime binding has also led to interest in **binding time flexibility**, whereby a variation point can be bound at different times [12, 19, 20]. The motivation is to maximize reuse of PLArtifacts across a larger variety of products. For instance, a middleware SPL could target both resource-constrained devices and high-end devices, and one variation point in this SPL could be the choice of a specific security protocol. For resource-constrained devices, small footprint size is more important than the flexibility of binding the variation point at runtime and thus the variation point is bound early with a specific security protocol. On the other hand, for high-end devices such flexibility is important and outweighs the incurred overhead (e.g., memory, performance loss due to indirection) of runtime binding of that variation point and thus the same variation point is bound at runtime depending on potential security threats or communication/throughput goals. Indeed, current research has proved the feasibility of implementing binding time flexibility, by using design patterns to make the variation points explicit and aspects to modularize binding time-specific code [12].

Although the relevance of binding time is well acknowledged [17], a concrete method for selection of the appropriate one and related to specific mechanisms is still missing. Such a method could leverage well-established practices in SPL and RTA, thus helping to explore their synergies.

Bindig time flexibility has increased the **importance of models** in RTA and SPL, e.g., DSPLs. For example, at earlier binding time, it is also important to model context in a more explicit and precise way, so that a *decison* about binding time can be made. Although acknowledged by traditional Domain Analysis, this has been represented informally and implicitly by the domain engineer. Conversely, in RTA, at later binding times, the decision model and context are still needed to decide on adaptation. Accordingly, for example, recent research also leverages the use of decision models at runtime [11]. Nevertheless, there remains the challenge of improving reasoning over this model at runtime. Conversely, the development of Reference Architecture in RTA could benefit from well established Domain Engineering approaches in SPL. This will help to discipline the process and leverage tested practices for building reusable artifacts. In particular, modern SPL component-based development processes such as Kobra [5] have features such as hierarchy model composition and refinement, and these could be enhanced with quality descriptions to be leveraged in RTA, thus helping to tame complexity.

The commonality among some models between SPL and RTA, the flexibility of binding time, and the blurredness of goals suggest that a holistic development process, exploring the synergies between SPL and RTA, would be beneficial to both domains. Particularly from the viewpoint of the RTA domain, there is still a general lack of appropriate engineering approaches. Accordingly, Adler et al. [2] introduced a classification with respect to the maturity of RTA approaches in three different evolution stages. In the state

of the practice, adaptation is usually used implicitly, without dedicated models at development time or even at runtime (evolution stage 1). In the current state of the art some approaches emerged which use distinct models for variability and decision modelling (evolution stage 2). This naturally helps coping with the high complexity of adaptive systems by making them manageable, i.e., by supporting the modular and hierarchical definition of adaptation enabling the model-based analysis, validation, and verification of dynamic adaptation. The existence of a dedicated methodology enabling developers to systematically develop adaptive systems is considered as a further evolution step (evolution stage 3).

A holistic model-based engineering approach would naturally also benefit from the whole range of typical gains brought by model-driven engineering (MDE) approaches (i.e. validation, verification, reuse, automation). As for any other software engineering approach it is particularly possible to analyze and to predict the quality of the adaptation behaviour to enable systematic control of the development process. In our opinion, the combination of SPL and RTA approaches could bear a significant step in this direction. Further, the benefits of the combination would also include more consistent handling of variability across the binding timeline and leverage of modelling and analysis techniques across both domains.

6. Related work

Indeed, describing potential synergy between variability in SPL and RTA is not new. For instance, each system configuration can be considered as a product in a SPL in which the variability decisions necessary to instantiate the product are made at run-time [25]. Cheng et al. [13] present a roadmap for engineering Self-Adaptive Systems, where they suggest that technologies like: model driven development, AOP, and SPL might offer new opportunities in the development of self-adaptive systems, and change the processes by which these systems are developed. In contrast to these works, we explore this synergy in the context of our concrete experience in the SPL and DAS domains and highlight some points that lead to further research.

Gokhale et al. [24] propose an initial approach for integrating Middleware with SPL, focusing on the use of feature-oriented programming and model-driven development tools for uncovering and exploring the algebraic structure of middleware and handling runtime issues such as QoS and resource management. Classen et al. [14] identify limitations in domain engineering in current SPL research and propose a research roadmap for the integration of SPL and RTA, based on the key concepts of context, binding time, and dynamism. Similarly to these works, we highlight QoS challenges and the role of models, in particular context and decision model; in contrast, we additionally discuss challenges regarding binding time flexibility.

The availability of decision models at runtime is regarded as an essential property of the synergy between SPL and RTA. Anastasopoulos et al. [4] investigate the benefits of applying SPL in the context of the Ambient Assisted Living domain [37], in which systems have to be highly adaptive, proposing a roadmap for its use. As in our work, they identify the need to focus on runtime variability and to provide an execution environment that enables management and automatic resolution of decision models at runtime. Their work additionally proposes corresponding realisation techniques and a component model. Cetina et al. [11] propose a method for developing pervasive applications using SPL concepts and techniques. The decision model is represented at runtime and queried during system reconfiguration in order to address new user goals. Differently, we also identify challenges on achieving binding time flexibility.

7. Conclusion

We performed a comparative study of variability management between SPL and RTA with the aim of identifying synergy points and cross-fertilization opportunities that could lead to enhanced variability management in both domains. Based upon meta models for each of the two domains and a set of general classification criteria, we identified and discussed potential synergy points. From a SPL point of view, potential synergies comprise the specification and management of QoS and dependability properties, a more systematic approach towards variable binding time, and the formalization of context information and its relation to product variants and their properties. From the perspective of RTA, well-established variability modelling in the SPL domain promises to be a valuable basis for the definition of appropriate models at runtime as they are required in adaptive systems. We believe that addressing these synergy points would be best exploited by the definition of a holistic model-based engineering approach, which we plan to refine in future work.

Although the comparison criteria used here are rather high-level, they are useful to structure the discussion and identify synergy points. Indeed, a more fine-grained comparison is needed, so that the two research areas can effectively benefit from each other. However, such a comparison is outside the scope this paper and is considered as future work.

Acknowledgements

The authors would like to thank Michalis Anastasopoulos, the anonymous reviewers, and the members of the SPG group at Federal University of Pernambuco for providing

valuable feedback. This research has been partially funded by the German Federal Ministry of Education and Research (BMBF) in the context of the VIERforES project.

References

[1] *SPLC'09. Call for Participation*. `http://www.sei.cmu.edu/splc2009/files/SPLC_2009_Call.pdf`. Last access, Nov. 2008.

[2] R. Adler, D. Schneider, and M. Trapp. Development of safe and reliable embedded systems using dynamic adaptation. In *1st Workshop on Model-driven Software Adaptation M-ADAPT'07 at ECOOP 2007*, pages 9–14, Berlin, 2007.

[3] M. Anastasopoulos and C. Gacek. Implementing product line variabilities. In *SSR '01: Proceedings of the 2001 symposium on Software reusability*, pages 109–117, New York, NY, USA, 2001. ACM.

[4] M. Anastasopoulos, T. Patzke, and M. Becker. Software product line technology for ambient intelligence applications. In *Proc. Net.ObjectDays*, pages 179–195, 2005.

[5] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based product line engineering with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[6] J. Bayer, O. Flege, and C. Gacek. Creating product line architectures. In *Third International Workshop on Software Architectures for Product Families - IWSAPF-3*, pages 197–203, 2000.

[7] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. Pulse: a methodology to develop software product lines. In *SSR '99: Proceedings of the 1999 symposium on Software reusability*, pages 122–131, New York, NY, USA, 1999. ACM.

[8] M. Becker, B. Decker, T. Patzke, and H. A. Syeda. *Runtime Adaptivity for AmI Systems - The Concept of Adaptivity*. IESE-Report; 091.05/E. 2005.

[9] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005.

[10] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair. Genie: Supporting the model driven development of reflective, component-based adaptive systems. In *ICSE 2008 - Formal Research Demonstrations Track*, 2008.

[11] C. Cetina, J. Fons, and V. Pelechano. Applying software product lines to build autonomic pervasive systems. In *SPLC '08: Proceedings of the 12th International on Software Product Line Conference*, pages 117–126. IEEE Computer Society, 2008.

[12] V. Chakravarthy, J. Regehr, and E. Eide. Edicts: implementing features with flexible binding times. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 108–119, New York, NY, USA, 2008. ACM.

[13] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors. *Software Engineering for Self-Adaptive Systems, 13.1. - 18.1.2008*, volume 08031 of *Dagstuhl Seminar Proceedings*. IBFI, Schloss Dagstuhl, Germany, 2008.

[14] A. Classen, A. Hubaux, F. Saneny, E. Truyeny, J. Vallejos, P. Costanza, W. D. Meuter, P. Heymans, and W. Joosen. Modelling variability in self-adaptive systems: Towards a research agenda. In *Proc. of the 1st Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering held as part of GPCE08*, October 2008.

[15] P. Clements and L. Northrop. *Software Product LinesPractices and Patterns*. Addison-Wesley, Reading, MA, 2002.

[16] C. F. Cortes, G. Blair, and P. Grace. An adaptive middleware to overcome service discovery heterogeneity in mobile ad hoc environments. *IEEE Distributed Systems Online*, 2007.

[17] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[18] A. Dey, D. Salber, and G. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction 16 (2)*, pages 97–166, 2001.

[19] E. Dolstra, G. Florijn, M. de Jonge, and E. Visser. Capturing timeline variability with transparent configuration environments. In *Proc. of International Workshop on Software Variability Management*, 2003.

[20] E. Dolstra, G. Florijn, and E. Visser. Timeline variability: The variability of binding time of variation points. In *Proc. of Workshop on Software Variability Management*, 2003.

[21] P. Fernandes and C. Werner. Ubifex: Modeling context-aware software product lines. In *Proc. of 2nd International Workshop on Dynamic Software Product Lines*, 2008.

[22] P. Fernandes, C. Werner, and L. G. P. Murta. Feature modeling for context-aware software product lines. In *SEKE*, pages 758–763, 2008.

[23] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin. Supporting adaptive multimedia applications through open bindings. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, page 128, Washington, DC, USA, 1998. IEEE Computer Society.

[24] A. Gokhale, A. Dabholkar, and S. Tambe. Towards a holistic approach for integrating middleware with software product lines research. In *Proc. of the 1st Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering held as part of GPCE08*, October 2008.

[25] H. Gomaa and M. Hussein. Model-based software design and adaptation. In *SEAMS '07: Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, page 7, Washington, DC, USA, 2007. IEEE Computer Society.

[26] P. Grace, G. Blair, C. Flores, and N. Bencomo. Engineering complex adaptations in highly heterogeneous distributed systems. In *Invited paper at the 2nd International Conference on Autonomic Computing and Communication Systems*, September 2008.

[27] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008.

[28] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using product line techniques to build adaptive systems. In *SPLC 2006: Proceedings of the 10th International Software Product Line Conference*, pages 141–150, Washington, DC, USA, 2006. IEEE Computer Society.

[29] D. Hughes, P. Greenwood, G. Coulson, G. Blair, F. Pappenberger, P. Smith, and K. Beven. An experiment with reflective middleware to support grid-based flood monitoring. *To appear in Wiley Inter-Science Journal on Concurrency and Computation: Practice and Experience*.

[30] D. Hughes, P. Greenwood, G. Coulson, G. Blair, F. Pappenberger, P. Smith, and K. Beven. An intelligent and adaptable flood monitoring and warning system. In *Proc. of the 5th UK E-Science All Hands Meeting (AHM06)*, 2006.

[31] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[32] C. W. Krueger. Product line binding times: What you don't know can hurt you. In *SPLC*, pages 305–306, 2004.

[33] J. Lee and K. C. Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *SPLC '06: Proceedings of the 10th International on Software Product Line Conference*, pages 131–140, Washington, DC, USA, 2006. IEEE Computer Society.

[34] G. Lenzini, A. Tokmakoff, and J. Muskens. Managing trustworthiness in component-based embedded systems. *Electron. Notes Theor. Comput. Sci.*, 179:143–155, 2007.

[35] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[36] D. Muthig. *A Lightweight Approach Facilitating an Evolutionary Transition Towards Software Product Lines*. Fraunhofer IRB Verlag, Stuttgart, 2002.

[37] J. Nehmer, M. Becker, A. Karshmer, and R. Lamm. Living assistance systems: an ambient intelligence approach. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 43–50, New York, NY, USA, 2006. ACM.

[38] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[39] C. P. Shelton, P. Koopman, and W. Nace. A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems. In *Words 2003: Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 156–163, Washington, DC, USA, 2003. IEEE Computer Society.

[40] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, 2005.

[41] M. Trapp. *Modeling the Adaptation Behavior of Adaptive Embedded Systems*. Verlag Dr. Hut, Munich, 2005.

[42] R. Wolfinger, S. Reiter, D. Dhungana, P. Grunbacher, and H. Prahofer. Supporting runtime system adaptation through product line engineering and plug-in techniques. In *Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008)*, pages 21 – 30, 2008.

[43] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, 2006. ACM.

Evolving a Software Product Line Reuse Infrastructure: A Configuration Management solution

Michail Anastasopoulos,
Dirk Muthig
Fraunhofer Institute for
Experimental Software Engineering
(IESE)
michail.anastasopoulos@iese.fraunhofer.de,
dirk.muthig@iese.fraunhofer.de

Thiago Henrique Burgos de Oliveira [1,2],
Eduardo Santana Almeida [1],
Silvio Romero de Lemos Meira [1,2]
[1]Recife Center for
Advanced Studies and systems - C.E.S.A.R
[2]Federal University of Pernambuco - UFPE
{thbo,esa,silvio}@cesar.org.br

Abstract

Configuration Management procedures are crucial for controlling the evolution of software products. In the context of Software Product Lines this becomes even more important since the development process consists of two parallel activities – Core asset and product development – and hence is more complex. This paper defines a set of Configuration Management procedures for Software Product Lines as well as an automation layer that facilitates their implementation.

1. Introduction

Product Line Engineering (PLE) is a paradigm on the rise that comes with true order-of-magnitude improvements in cost, schedule and quality [3]. For achieving these improvements it is necessary that an initial investment is made in terms of establishing systematic reuse. Naturally every organization expects that such an investment (a) amortizes quickly and (b) persists during the evolution of a product line.

Amortization is supported by solutions, such as the approach in [1], that address the proactive planning of product line reuse, so that expected benefits and their timing are clarified in advance. Or, there are techniques [2] that support the development for reuse in a product line context so that reusable artifacts are easier to implement and reuse. However, when it comes to evolution of a product line over the time, there is currently no consensus regarding the approach to be followed.

1.1. Problem Statement

Currently, there is no established solution that addresses all three essential product line activities identified in [3]. These activities are:

- Core Asset Development (also known as Family Engineering): In this activity the core assets are being developed and evolved, that is the assets to be reused across the different members of the product line.
- Product Development (also known as Application Engineering): In this activity core assets are reused, that means that so-called core asset instances are being derived and then evolved. In addition product-specific assets are being developed. Reuse entails configuration, derivation and finally adaptation. The resulting core asset instances together with product-specific assets form the products to be delivered to customers.
- Management: This involves organizational and technical management. A major issue here is the definition and monitoring of the processes used for developing core assets and products. This activity is crucial for evolution management.

A recent analysis of several industrial product lines [4] has shown that in most cases Core Asset and Product Development are separate activities with separate lifecycles, which run in parallel. If the interactions between these activities are not enforced and properly managed the risk of product line "decay" grows [19, 20, 24]. The latter can be defined as the aggravating situation, in which core asset reuse diminishes while the amount of product-specific assets grows. Such a situation can have several negative side-effects including the exponential growth of maintenance effort over time and finally the failure of a product line effort.

For the avoidance of product line decay, special attention must be paid to the *Management* essential activity. This paper provides solutions that directly contribute to that.

1.2. Solution Approach

The solution presented in the following is based on configuration management, which is an established management discipline for controlling the evolution of software systems. Since traditional configuration management does not address special aspects pertaining to product lines, a set of extensions are proposed to fill the respective gaps. The solution is made up of two components.

- RiPLE-EM: A process model that describes the roles, activities and flows that are necessary for a disciplined product line evolution management.
- Customization Layer: An automation layer that (a) defines basic product line evolution activities, (b) defines an asset model for the management of core asset / instance dependencies and (c) automates many of the manual steps that would be otherwise necessary if plain configuration management was used.

As shown in Figure 1 RiPLE-EM builds on top of the Customization Layer thereby using and coordinating the basic product line evolution activities defined therein. The Customization Layer in turn builds on a Configuration Management Repository and in particular on the available version control functionality.

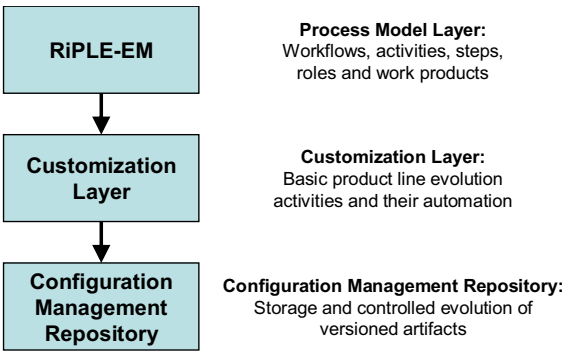


Figure 1: Solution Layers

The Customization Layer is part of the PuLSE™ method (Product Line System and Software Engineering)[1] and in particular it belongs to the PuLSE™-EM technical component (Evolution and Management), which also defines a process model similar to RiPLE-EM. The latter takes however a more configuration management-oriented view, by also addressing issues of Configuration Identification, Release and Build Management. Hence this paper aims at bringing together the RiPLE-EM and the PuLSE™ views.

1.3. Paper structure

The paper is structured as follows: section 2 presents related work, Section 3 presents RiPLE-EM and section 4 the underlying Customization Layer. Section 5 discusses the interfaces between the two and section 6 closes the paper.

2. Related Work

Related work on evolution management for product lines can be separated in two areas: (a) related work on process models and (b) related work on technologies. Normally process models and technologies should be closely related. Process models should be genuine reflections of the evolution processes that are supported by technologies. However such a contribution is not yet available in the community and therefore the separate observation of the two areas is necessary.

2.1. Process Models

Besides PuLSE-EM, which was discussed in section 1.2, there are several other process models for PLE such as KobrA [5] and FAST [6] but as mentioned in the introduction, there is no established solution regarding evolution management processes. Most of the existing process models address PLE, but do not provide any detailed activity flow for conducting and coordinating evolution activities.

The research projects ESAPS [21] and CAFÉ [22] also discussed evolution management approaches mainly in terms of change management and change impact analysis. Yet they do not provide any kind of workflow, or suggested activities. The process model proposed by this paper differs from these existing approaches by defining workflows with activities, steps to accomplish each activity, roles and work products. Apart from that RiPLE-EM also provides concrete process guidelines to some of its activities.

Some isolated guidelines on how to conduct evolution management activities and procedures are available in different papers, mainly in the form of experience reports such as [23] and [24]. However, there is no compilation of these guidelines and best practices in the form of a process model definition, which helps in enabling the coordination of these

[1] PuLSE™ is a registered trademark of the Fraunhofer IESE) in Kaiserslautern, Germany www.iese.fraunhofer.de/Products_Services/pulse/

activities in a uniform way, and providing a common understanding of the proposed activities execution.

2.2. Technologies

Among the different product line implementation technologies there are currently two major tool-based solutions that explicitly discuss the evolution of product lines. However these technologies do not address the full-fledged product line scenario characterized by the interplay of core asset and product development.

The first solution is available through the GEARS software production line [7]. The latter is defined [8] as a special kind of product line, in which product development is not in the focus of the product line infrastructure. GEARS provides the methodology and tools for the specification of variability as well for the definition, realization and instantiation of the respective core assets. However, once generated, the instances of core assets are transient work products and GEARS does not deal with their evolution. If changes are necessary they are performed only within the infrastructure. Instances can then be re-generated. GEARS is an interesting solution when the separation of core asset and product development is not necessary or feasible. This can be the case in smaller or medium-sized organizations. Apart from that, and although GEARS supports integration with various configuration management systems, the mechanisms of storing and retrieving versioned data have not been published yet. The approach presented in this paper automates in a well-defined way a series of configuration management operations for controlling core assets and instances.

GEARS falls into the category of variability management environments. The Decision Modeler [2], pure:variants [10] or COVAMOF [11] are comparable solutions. Nevertheless those solutions do not address the evolution issues explicitly and therefore they can be used only in conjunction with a configuration management solution (more on that in section 4.6)

The second solution comes with the KOALA component model and according infrastructure [9]. This solution also addresses a special kind of product line termed product population. In this case the reuse infrastructure consists of KOALA components, which are highly reusable core assets according to the principles of component-based development [12]. A product based on KOALA is derived through composition of KOALA components. Since the derived products can be very dissimilar one speaks of product populations instead of product lines. Hence KOALA components can be seen both as core assets and instances depending on the timing of component composition. KOALA does not need to address the issue of coordination between core asset (i.e. component) development and product development (i.e. component composition). That is due to the fact that the application of component-based development principles in KOALA achieves a very high reuse level. The latter makes the adaptation of components unnecessary. Therefore change requests from the products can be directly propagated to the component development thereby yielding new versions of components. KOALA is therefore a very powerful approach. However it can be applied only at the component design and implementation level. This is the difference to the solution presented in this paper, where the management of core assets in general (i.e. pertaining to other lifecycle phases as well) is supported.

Apart from GEARS and KOALA, which are modern state-of-the-practice approaches in the product line community, there is also a set of research prototypes from the configuration management community that, although address evolution issues of product lines, did not fully make it to industrial practice. VOODOO [13], ICE [14], Adele [15] and the ShapeTools [16] are representative examples. The major enhancements of the solution presented in this paper against these approaches are (a) the support for the coordination between core asset and product development as well as (b) the provision of a product line evolution front end on top of an encapsulated configuration management system.

A research prototype with the most similarities to the automation solution of this paper is the Product Line Asset Manager [17]. The latter addresses most of the issues discussed in section 1 including the definition of roles, processes and models for product line evolution management. The automation solution of this paper takes a similar approach but delves deeper into the instrumentation of the underlying configuration management system and into the definition and usage of the asset model, which shows the relationships between core assets and instances.

3. RiPLE-EM

In this section, the process model for evolution management, called RiPLE-EM (RiSE[2] Product Line Engineering – Evolution Management) is presented. For the reason of simplicity, we use the word "process" instead of "process model" in the following. RiPLE-EM process specifies roles, activities and flows

[2] RiSE – http://www.rise.com.br

required for the management of evolution in a product line context.

3.1. Motivation

Processes are an important aspect of evolution management. In the product line context, where the interplay between Core Asset and Product development increases the complexity, process models are necessary to give concrete guidance to the product line stakeholders. RiPLE-EM aims at providing a semi-formal process that (a) integrates and coordinates the Customization Layer operations and that (b) surpasses that, by defining further activities for configuration identification, change management, build management and release management. The latter are fundamental functions of traditional configuration management. Hence RiPLE-EM adapts these functions to the context of product lines.

3.2. Process Model

The main characteristic of RiPLE-EM is that it is focused both on core asset and product development, in a release-oriented way. By release oriented, we mean that for each core asset or product release, a new RiPLE-EM flow is started and followed.

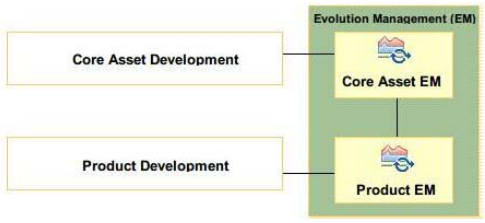


Figure 2: RiPLE-EM overview

RiPLE-EM has two different flows, one for each of the two essential activities of Core Asset Development (CAD) and Product Development (PD), discussed in section 1. Figure 2 gives an overview of RiPLE-EM.

RiPLE-EM consists of the following sub-areas, each one with different activities, described in the following subsections.

- **Configuration Identification**: This area identifies the artifacts to be put under evolution control. It is decided which artifacts will be developed as core assets, instances, or product specific assets.
- **Change Management**: This sub-area comprises all activities related to the change control of the configuration items. It comprises activities like requesting and analyzing changes and propagations.
- **Build Management**: All activities related to build generation are under this sub-area, which defines different types of builds (private system build, integration build and release build) and provides the steps to accomplish the build generation, taking in consideration variability and configuration matters.
- **Release Management**: This sub-area comprises both release planning and execution for core assets and products.
- **Support activities**: Further activities such as audits and status reporting.

Figure 3 illustrates the usage of RiPLE-EM in a given moment when two core assets and two products are being developed. For the proper management of the evolution, each release (of the core asset or the product) follows the RiPLE-EM respective flow (CAD for core assets and PD for products).

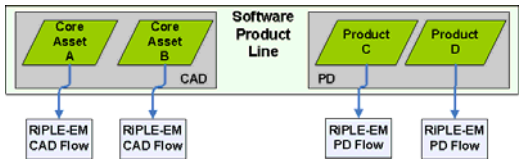


Figure 3: RiPLE-EM usage example

3.2.1. Core Asset and Product Development

As mentioned before RiPLE-EM is separated in two different flows, the RiPLE-EM for Core Asset Development (RiPLE-EM CAD or simply CAD) and the RiPLE-EM for Product Development (RiPLE-EM PD or simply PD). These two flows have similar macro structure, but the activities inside each flow have different goals.

RiPLE-CAD focuses on the proper evolution and release of core assets, including the identification of items and the release planning of a certain core asset, change management of those items, build management when applicable (e.g. code development), and release execution practices where the core asset is made available for the product development.

The focus of RiPLE-EM PD is different from RiPLE-EM CAD. In the configuration identification, and release planning, assets to be included in the products are identified and tracked. It has a different focus in the change management, where assets can be added, or instantiated from existing core assets; in the build management, where the product is built, and all variability decisions resolved; and also in the release execution, where the product is released and published and the changes performed in the product release can be propagated back to core assets base.

The communication of CAD and PD is also considered, in both directions, in terms of change propagation. The change propagation request (PR) is

the instrument used to initiate and control the propagation of changes between core asset and product development by the management team.

3.2.2. RiPLE-EM for Core Asset Development

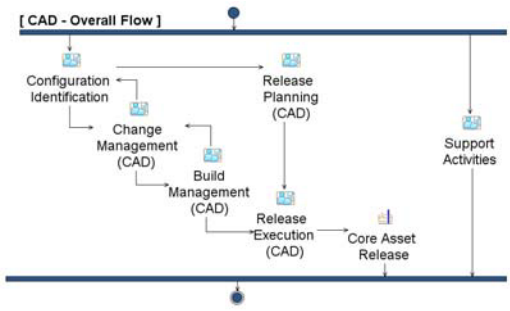


Figure 4: RiPLE-EM CAD flow

As shown in Figure 4, the flow starts with the configuration identification of the core asset to be developed. The assets can be of different types, such as components, architecture, frameworks, requirements and etc. The configuration identification of the items that will be developed inside the core asset is followed by the development itself, supported by the change and build management activities. The flow finishes with the release execution of the core asset. The release planning starts right after the configuration identification and continues by monitoring the development and updating the release plan. Finally it joins the development flow at the moment of the release execution[3]. There are also support activities such as audits that can be performed at any time during the RiPLE-EM CAD process.

Inside the change management, RiPLE-EM CAD supports the communication with product development through propagation requests (i.e. rebase core asset instances) that can be raised and further analyzed by the responsible roles. Variability changes (i.e. addition or removal of variation point) are also handled by a specific task where guidelines for the change are proposed. Moreover every change or propagation request records the rationale for it, providing shared knowledge. The build management of a core asset is also impacted by the variability points and options available inside the core asset, enabling the generation of pre-configured core asset builds.

Inside each of the CAD activities there are further flows and documentation regarding the steps to accomplish every task, as well as the artifacts, guidelines and roles associated.

3.2.3. RiPLE-EM for Product Development

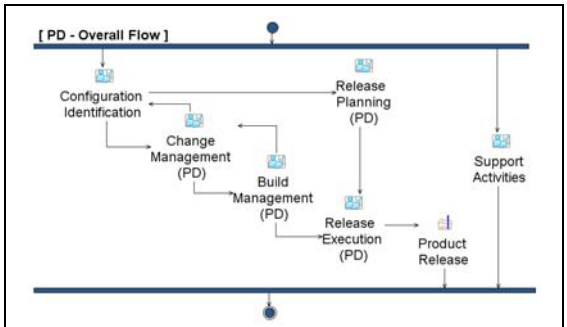


Figure 5: RiPLE-EM PD flow

RiPLE-EM for PD, as shown in Figure 5, also starts with the configuration identification and the release planning for the product. In the latter the release schedule and schedule dependencies are defined (e.g. Core assets that are still being developed). The development activity is again supported by change and builds management. The latter enable the creation of product specific assets, or core asset instances for the product being developed. In the release execution of a product, activities such as the creation of a release notes, publication and eventual propagation are supported by the process.

4. Customization Layer

This section presents the Customization Layer that provides a set of basic product line evolution operations by combining functions of traditional configuration management.

4.1. Motivation

Product line engineering is characterized by systematic reuse, which in turn requires variability management. Hence when a product line is to be evolved with traditional configuration management, means have to be found for dealing with variability. Since the configuration management spectrum of functionality is quite broad, there are several solutions that support variability management. The typical solution is provided by the branching and merging functionality. However it is recognized [16] that traditional branching and merging is not adequate for dealing with permanent variations.

The main problem is that the increased numbers of variations and variation types lead to many interdependent branches [13]. Although configuration management systems have normally no problems in

[3] Normally this would require another set of fork and join nodes in the workflow diagram; however this was left out for simplicity. The same applies to Figure 5.

dealing with a high number of branches, it is the user that can get easily overwhelmed. This can be due to various situations such as (a) usage of branches for purposes other than variability management, (b) inconsistent branch naming or (c) formation of totally dissimilar revision graphs for the same kind of variability. A user that wants to get information – stored in branches – about variability and evolution has to filter out all irrelevant data that comes out from the above situations. And for doing that a user has to combine information from log files, revision graphs and the repository layout. Hence the complexity the user is confronted with grows rapidly.

There are also other configuration management solutions to the variability problem, for example through the change and build management functionalities. However such solutions do not fully solve the problem. Change management can indeed define the roles, activities and flows necessary for dealing with changes in a product line. This is exactly what the RiPLE-EM described in section 3.2 does. Yet, the enactment of some of these activities entails the difficulties discussed in the beginning of this section. Build management (e.g. make files) on the other hand enables the implementation of product line configurators. In this case the derivation of products is well supported however evolution is not addressed.

Finally, another conceivable solution would be the implementation of new kind of configuration management system (for example with the help of a database management system as it is frequently the case) that fits exactly the needs of product line engineering. Yet the adoption of such a solution could be questionable since organizations have typically invested a significant amount of effort and budget in acquiring, learning and using a standard configuration management system.

Hence in this paper we propose a solution that closes the gap between product line engineering and traditional configuration management through abstraction and encapsulation. Abstraction enables the definition of a product line evolution front-end with the evolution operations needed in product line engineering, and encapsulation provides for the implementation of these operations through automated combination of low-level configuration management primitives.

4.2. Product Line Evolution Elements

Figure 6 illustrates the fundamental elements of product line evolution management and their relations. In other words the figure shows the concepts a product line engineer (i.e. core asset or product engineer) uses during evolution of a product line:

- Core Asset: A core asset is defined as a reusable asset that is developed in Core Asset development and reused many times during product development. A core asset may contain other core assets.
- Instance: An instance is defined as copy of a core asset that is reused (i.e. configured and adapted) in the context of a product during product development. An instance may also contain other instances.
- Instantiation: An instantiation is defined as the act of resolving (fully or partially) the variation points contained in a core asset thereby yielding an instance of the core asset that can be then adapted in a product-specific way.

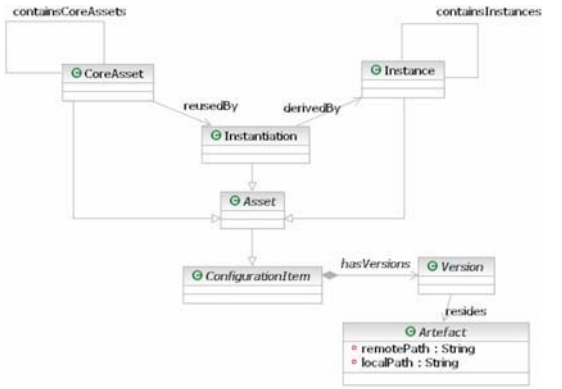


Figure 6: Elements of product line evolution

As shown in Figure 6 every core asset and every instance are assets of the product line and that means that they are configuration items that can be evolved (i.e. versioned) over time. Moreover they have to encapsulate a physical artifact, a file or a directory.

Figure 6 also implies that every instantiation is an asset and hence a configuration item as well. The idea is (a) that an instantiation should describe what the product engineer did when he derived an instance and (b) that the way a given instance is derived from a core asset may change over time. This happens when the variation points of core assets change. For example a variation point that allowed three alternatives may evolve to allow only two alternatives. In such a case the existing instances of the core asset may have to be revisited and eventually re-instantiated. The latter leads to a new version of the according instantiation object. Finally a new version of the according instance is created. In this way the model enhances the consistency between core assets, instantiations and instances over time.

The model presented in Figure 6 can be nicely integrated with variability management models (e.g.

[25] or [26]) that introduce concepts such as variation point, decision, constraint etc. Yet this exceeds the context of this paper.

4.3. Product Line Evolution Operations

The model of Figure 6 presents the basic elements of product line evolution. The Customization Layer provides the operations necessary for creating and manipulating instances of this model.

The operations currently supported by the Customization Layer are the following. For simplicity the detailed signatures are left out at this point.

1. add-core-asset: Create a core asset from an artifact (file or directory) and add it to the configuration management repository.
2. show-core-assets: Given a location in the configuration management repository (or the location defined as standard if none is passed), it shows all core assets.
3. show-instance-diff: Given a core asset, check whether its instances have changed since their derivation from the core asset
4. integrate: Given a core asset and one of its instances, mark the last change made to the core asset as a feedback from the instance to the core asset
5. instantiate-core-asset: Given a core asset create an instance of the core asset. The instance is basically a copy of the core asset where all or a part of the core asset variation is resolved. The Customization Layer however does not enforce resolving any variability. It simply creates a copy and assumes that some kind of development with reuse takes place during the copy operation or afterwards.
6. show-instances: Given a core asset or the complete product line, show the current instances
7. show-core-diff: Given an instance, check whether its core asset has changed since the derivation of the instance.
8. rebase: Given an instance mark the last changes made to the instance as a feed-forward from the core asset of the instance to the instance itself.

Commands 1 to 4 are meant for usage by Core Asset Engineers while commands 5 to 8 are meant for Product Engineers.

4.4. Configuration Management

For realizing the operations of section 4.3 the Customization Layer encapsulates standard configuration management functionality. The usage of standard functionality has been chosen intentionally for enabling the usage of the Customization Layer with any configuration management system. Clearly, there are features of some configuration management systems that could support product line evolution better. Such features will be discussed in the following two sections. Apart from the fact that these features do not solve all problems, taking them for granted would bind the solution to a specific configuration management system.

4.4.1. Centralized Configuration Management

Centralized configuration management systems provide a central repository where the history of all artifacts is maintained. There are features of such systems that could be beneficial in a product line context. For example ClearCase [18] supports a special kind of branches, called integration and development streams. Core asset development could be made with an integration stream whereas Product Development with various development streams. ClearCase would then facilitate the propagation of changes. This however does not solve the problem of many different types of streams, their relations and the changes among them than have to be monitored and controlled. Another example would be Subversion [28] that enables the assignment of custom properties to configuration items and also enables the a-posteriori modification of commit messages. Such features could be used for marking items as core assets or as instances or for enriching commit messages with integration or rebase comments (see section 4.3)

4.4.2. Distributed Configuration Management

Distributed version control systems like Git [27] or Mercurial [29], take a decentralized approach: Every user maintains its own repository and changes can be propagated between repositories over the network. In a product line context such a distribution of repositories can be beneficial. Similarly to ClearCase streams there could be a repository for the core asset development and several other repositories for the development of various products. Subsequently these repositories can push or pull changes among them. Also in this case the question remains on how to manage the relations and change events between the different repositories.

4.4.3. Encapsulation of Standard Functionality

The configuration management functionality required by the Customization Layer is the following:

- Creating configuration items: The Customization Layer uses this feature when the add-core-asset command is used. The result is a new configuration item with the contents of the core asset and the repository location passed to the command. A specific commit message is used every time for separating such a configuration

items from others created differently. The Customization Layer creates some configuration items also during initialization. These are the predefined repository directories the Customization Layer uses as root directories for core assets and instances respectively.

- Branching off configuration items: Branches are used with the instantiate-core-asset command. When this command is issued on a core asset the Customization Layer creates a branch off the core asset and hence enables the parallel development of the core asset and the newly created instance.
- Searching the history of configuration items: This functionality is used with the majority of the Customization Layer commands. For example when show-core-assets is issued the Customization Layer searches the history of the repository location passed to the command for entries bearing the special commit message used by the Customization Layer. Or when show-core-diff is called the history is repeatedly searched for comparing version numbers and deducing whether a change has happened.
- Creating versions: This functionality is used during integrating and rebasing. In each case special-purpose versions are created that let the Customization Layer later deduce that this change was a feedback or a feed-forward respectively.

4.5. Customization Layer Usage Scenarios

The Customization Layer can be used for two scenarios. The primary scenario is the controlled evolution of core assets, instances and their dependencies. With the Customization Layer product line engineers can continuously keep an overview over all core assets, instances and changes taking place therein. Yet the Customization Layer does not support the semantic interpretation of changes. The latter can be very challenging in a product line context since core assets and instances cannot be easily compared. For minimizing the effort of the interpretation the Customization Layer enables finding out quickly that a change happened so that an analysis can be initiated early.

The second scenario the Customization Layer can be used for is development for reuse. Typically core assets are made reusable through low-level reuse mechanisms of programming languages (e.g. template meta-programming, preprocessor directives, class inheritance) or high-level mechanisms of component technologies (e.g. parameterization, dependency management, extension points). Configuration Management can be also used for development for reuse; however in a rather primitive way.

For example a core asset supports two operating systems (Windows and Linux). The core asset engineer may decide to implement this core asset as a directory that has two branches, the Windows (e.g. as the main branch) and the Linux branch. During instantiation the application engineer would select one of the branches (configuration step), check-out the latest revision in the branch (derivation step) and then introduce his product-specific changes (adaptation step). The Customization Layer can support this scenario: instantiate-core-asset can be called on the core asset module. That would create a copy of the module with the two branches and their latest revisions. Subsequently this copy, which makes up the instance of the core asset, can be "configured" by deleting for example the Windows branch, if a Linux instance is required.

4.6. Interaction with Variability Management

Examining the relation between the Customization Layer and a variability management technology like GEARS, the Decision Modeler or pure:variants is important. While Customization Layer focuses on the evolution of products variability management technologies focus on the derivation of products. With variability management a domain space is typically built-up that specifies the decisions an application engineer has to make when deriving a product or parts of a product. Decisions in the domain space are mapped to logic that delivers the assets accordingly. Hence when a product is to be derived a set of decisions are being made and then the according logic is executed that selects, generates, transforms, deletes etc. assets. The result is the assets that make up a product or parts of a product.

The first possible interaction between the Customization Layer and variability management is hence the domain space and the logic creation step. Whenever a core asset is defined within the variability management environment the add-core-asset command can be called for storing the core asset accordingly in the configuration management repository.

The second possible interaction is the derivation step. During such an interaction the variability management calls the Customization Layer. For each core asset being processed by the logic it calls the instantiate-core-asset command with the core asset as first argument and the instance as second argument. The second argument is provided if an instance is delivered by the variability management (e.g. through automated transformation of the core asset). If no second argument is provided the Customization Layer

creates a copy of the core asset and marks that accordingly. The result is that after product derivation the Customization Layer has stored instances in a controlled way and evolution monitoring can then start.

Finally the Customization Layer can also call a variability management environment in case core assets or instances change. In that way the variability management environment can automatically change its domain space or logic. If for example an instance is created only with the Customization Layer the variability management environment could be notified so that the domain space and the according logic can be revisited.

Given the example in the previous paragraph it is also conceivable that the Customization Layer is used in a stand-alone mode; that is without any interaction with a variability management environment. Indeed the Customization provides basic means for variability management by enabling the creation and overview over core assets and instances. However it does not provide any means for logical dependencies, in terms for example of constraints, between core assets. This is typical application field of variability management environments. Hence the interaction between Customization Layer and a variability management environment is highly recommended.

5. Interfaces between RiPLE-EM and CL

To address a complete solution regarding evolution management in Software Product Lines, RiPLE-EM and the Customization Layer have integration points where both solutions can be combined to maximize the benefits. Each Customization Layer operation, listed in section 4.3, is triggered by a RiPLE-EM task or activity. The interfaces between the solutions are described next, and summarized in Figure 7.

- **add-core-asset**: This situation may occur in the change management flow, from both CAD and PD, when the need for a core asset creation is identified through change request analysis or through the configuration identification.
- **show-core-assets**: The need to visualize the available core assets may arise from the activity of identifying the configuration of a certain product which will re-use the core assets, from the instantiation of a core asset or from the propagation of changes.
- **show-instance-diff**: Any time a change propagation request is to be opened, it is important to know beforehand the changes made in the instance, in order to verify the applicability of the propagation. This operation is also triggered in the analysis of a propagation request.
- **integrate**: Every time the propagation is realized, this operation is triggered to mark that the core asset was update with changes from a specific instance.
- **instantiate-core-asset**: When an instance needs to be created during PD configuration identification or PD change management this operation is triggered.
- **show-instances**: Given a certain core asset, it is always interesting to know which product have an instance of that asset, specially for the purpose of requesting propagation or simply analyzing a certain change request.
- **show-core-diff**: For product engineers, having the possibility of knowing when the base core asset for a given instance changed, to rebase it.
- **rebase**: Similar to the integrate operation. When the change propagation is realized, this operation is triggered to mark that the instance base was updated with changes from the core asset derived.

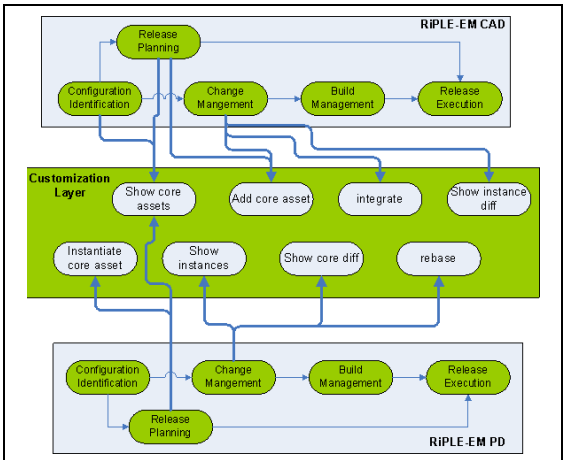


Figure 7: RiPLE-EM and CL integration

6. Conclusions

The parallel execution of Core Asset and Product development activities makes the evolution management in software product lines much more complicated than in single systems. A mature and established technology like configuration management can provide good support in that direction. Yet traditional configuration management does not match the particulars of product lines and its direct usage in a product line context becomes complicated. In this report the Customization Layer has been presented, an automation layer on top of traditional configuration management that provides the evolution operations

that concern product line engineers but hides away the underlying complexity.

The report also presented RiPLE-EM, a process model that in turn sits on top of the Customization Layer coordinating the activities provided therein and providing a full set of roles, activities and flows guiding product line evolution.

Future work in the Customization Layer includes thorough validations, the incorporation of additional operations such as structuring (e.g. defining logical hierarchies of core assets – mapping to the software architecture), removing (e.g. deleting an instance) or shifting (e.g. make a core asset out of an instance) operations and finally the better enforcement of policies. As for the RiPLE-EM, future work includes the tailoring of the process to offer better support to different product line directions (e.g. production lines, product populations as discussed in section 2.2), as well as the tailoring for an agile version of RiPLE-EM.

7. References

1. John, Isabel ; Barreto Villela, Karina: Evolutionary Product Line Requirements Engineering 12th International Software Product Line Conference, IEEE Computer Society, 2008, 374-375
2. Yoshimura, Kentaro et al: Model-based Design of Product Line Components in the Automotive Domain 12th International Software Product Line Conference, IEEE Computer Society, 2008, 170-179
3. Clements, Paul; Northrop, Linda: Software Product Lines. Practices and Patterns Boston: Addison-Wesley, 2002.
4. Linden, Frank van der ; Schmid, Klaus ; Rommes, Eelco: Software Product Lines in Action : The Best Industrial Practice in Product Line Engineering Berlin : Springer-Verlag, 2007, section 18.4
5. Atkinson, Colin et al, Component-based Product Line Engineering with UML: Addison-Wesley, 2001.
6. Weiss, David M.; Lai, Chi Tau Robert: Software Product-Line Engineering. A Family-Based Software Development Process Reading : Addison-Wesley, 1999
7. GEARS Homepage, retrieved November 2008 from www.biglever.com/solution/solution.html
8. Krueger, C. W. 2002. Variation Management for Software Production Lines. Second international Conference on Software Product Lines (2002)..
9. van Ommering, R.; van der Linden, F.; Kramer, J.; Magee, J., "The Koala component model for consumer lectronics software," Computer , vol.33, Mar 2000
10. Beuche, D. 2008. Modeling and Building Software Product Lines with Pure::Variants. 12th international Software Product Line Conference IEEE Computer Society, Washington, DC, 358
11. M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch, "Modeling Dependencies in Product Families with COVAMOF," in Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, 2006, pp. 299-307
12. Szyperski, Clemens:Component Software. Beyond Object-Oriented Programming 2nd ed. London : Addison-Wesley, 2002
13. Reichenberger, C. 1995. VOODOO - A Tool for Orthogonal Version Management. Lecture Notes In Computer Science, vol. 1005. Springer-Verlag, London, 61-79.
14. Zeller, A. and Snelting, G. 1997. Unified versioning through feature logic. ACM Trans. Softw. Eng. Methodol. 6, 4 (Oct. 1997), 398-441.
15. Estublier, J. 1995. The Adele configuration manager. In Configuration Management, W. F. Tichy, Ed. Wiley Trends In Software Series, vol. 2. John Wiley & Sons, New York, NY, 99-133.
16. Mahler, A. 1995. Variants: keeping things together and telling them apart. In Configuration Management, W. F. Tichy, Ed. Wiley Trends In Software Series, vol. 2. John Wiley & Sons, New York, NY, 73-97.
17. Stefan Bellon and Jörg Czeranski, Thomas Eisenbarth, Daniel Simon, A Product Line Asset Management Tool, 2nd Groningen Workshop on Software Variability Management Groningen, Netherlands, December 2004
18. Homepage of Rational ClearCase retrieved November 2008 from www.ibm.com/software/awdtools/clearcase/
19. Liguo Yu and Srini Ramaswamy, "A Configuration Management Model for Software Product Line," INFOCOMP Journal of Computer Science Vol. 5, No. 4, December 2006, pp. 1-8.
20. Davis, A. M. and Bersoff, E. H. 1991. Impacts of life cycle models on software configuration management. Commun. ACM 34, 8 (Aug. 1991), 104-118
21. Homepage of ESAPS, retrieved November 2008 from http://www.esi.es/esaps
22. Homepage of CAFÉ, retrieved November 2008 from http://www.esi.es/Cafe
23. R. Kurmann. Agile software product line configuration and release management. Workshop on Agile Product Line Engineering in the Software Product Line Conference, 2006
24. Staples, M.; Change control for product line software engineering, Software Engineering Conference, 2004. 11th Asia-Pacific 30 Nov.-3 Dec. 2004 Page(s):572 – 573
25. Becker, Martin: Anpassungsunterstützung in Software-Produktfamilien Kaiserslautern : Technische Universität Kaiserslautern, 2004. (Schriftenreihe / Fachbereich Informatik, Technische Universität Kaiserslautern; Bd. 19).
26. Muthig, Dirk: A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines Stuttgart : Fraunhofer IRB Verlag, 2002. (PhD Theses in Experimental Software Engineering; Vol. 11).
27. Homepage of git, retrieved December 2008 from http://git-scm.com/
28. Homepage of Subversion, retrieved December 2008 from http://subversion.tigris.org/
29. Homepage of mercurial, retrieved December 2008 from http://selenic.com/mercurial/

Analysis of Feature Models using Generalised Feature Trees

Pim van den Broek
Department of Computer Science,
University of Twente
P.O. Box 217,
7500 AE Enschede,
The Netherlands
pimvdb@ewi.utwente.nl

Ismênia Galvão
Department of Computer Science,
University of Twente
P.O. Box 217,
7500 AE Enschede,
The Netherlands
i.galvao@ewi.utwente.nl

Abstract

This paper introduces the concept of generalised feature trees, which are feature trees where features can have multiple occurrences. It is shown how an important class of feature models can be transformed into generalised feature trees. We present algorithms which, after transforming a feature model to a generalised feature tree, compute properties of the corresponding software product line. We discuss the computational complexity of these algorithms and provide executable specifications in the functional programming language Miranda.

1. Introduction

Feature models are used to specify the variability of software product lines [1,2]. To calculate properties of software product lines which are specified by feature models, such as the existence of products, a number of approaches exist in the literature where feature models are mapped to other data structures: Benavides et al. [3] use Constraint Satisfaction Problems, Batory [4] uses Logic Truth Maintenance Systems and Satisfiability Solvers, and Czarnecki and Kim [5] use Binary Decision Diagrams.

The decision problem to determine whether a feature model has products is, in general, NP-complete. The mappings to Constraint Satisfaction Problems and Logic Truth Maintenance Systems can be performed in polynomial time, but the resulting problem is also NP-complete. Although with Binary Decision Diagrams the problem only requires constant time, the mapping from feature models to Binary Decision Diagrams takes exponential time in the worst case.

In a previous paper [6] we have shown how feature models which consist of a feature tree and additional constraints can be transformed into trees. Although this transformation takes exponential time in the worst case as well, it is feasible when the number of constraints is small. The resulting trees are more general than feature trees, since features may have multiple occurrences. In this paper we study a special subset of those trees, called *generalised feature trees*, and show how they can be used to compute properties of the corresponding software product lines.

In the next section we briefly describe the feature models we consider in this paper. In section 3 we introduce the concept of generalised feature tree and describe algorithms which deal with commitment to a feature and deletion of a feature of a GFT. In section 4 we describe how a large class of feature models can be mapped to equivalent GFTs. In section 5 we show how this mapping can be used for the analysis of feature models. In section 6 we present an example and in section 7 we discuss the computational complexity of our approach. Throughout the paper, we present executable specifications of all algorithms in the functional programming language Miranda [8].

2. Feature models

The feature models we consider in this paper consist of a feature tree and a set of constraints. A feature tree is a tree which can have three kinds of nodes: MandOpt nodes, Or nodes and Xor nodes.

A MandOpt node has two sets of child nodes, called mandatory and optional nodes respectively. Or nodes and Xor nodes have 2 or more child nodes. A leaf of the tree is a MandOpt node without children. Just for the

ease of writing concise algorithms, we assume the existence of a special feature tree NIL, which has no nodes. Each node of a tree has a feature, which is just a list of characters. All nodes in a feature tree have different features, and NIL does not occur as subtree of any feature tree. A product is a set of features. A constraint maps products to Boolean values; in our prototype implementation the constraints are restricted to constraints of the forms "A requires B" and "A excludes B".

In Miranda, these type definitions are as follows:

```
tree ::= MandOpt feature [tree] [tree] |
         Or feature [tree] |
         Xor feature [tree] |
         NIL
feature == [char]
product == [feature]
constraint::= Requires feature feature |
              Excludes feature feature
feature_model == (tree,[constraint])
```

The semantics of a feature model is a set of products [7]; it consists of those products which satisfy the constraints from the tree as well as the explicit constraints.

A product satisfies the constraints from the tree if:

- All its features are features of a node in the tree.
- It contains the feature of the root of the tree.
- For each feature of a node n in the product: if n is not the root, then the product contains also the feature of the parent node of n.
- For each feature of a MandOpt node in the product, the product also contains all features of its mandatory child nodes.
- For each feature of an Or node in the product, the product also contains one or more of the features of its child nodes.
- For each feature of an Xor node in the product, the product also contains exactly one of the features of its child nodes.

A product satisfies a constraint "A requires B" when, if it contains A it also contains B. A product satisfies a constraint "A excludes B" when it does not contain both A and B.

3. Generalised feature trees

Features in a feature tree are, albeit implicitly, required to be all distinct. In the generalisation of feature trees we consider in this paper, this requirement is somewhat relaxed. We define a generalised feature tre (GFT) to be a feature tree whose features, instead of being required to be all distinct, satisfy the following two restrictions:

- Restriction 1: when two nodes of a GFT have the same feature, they belong to different subtrees of an Xor node.
- Restriction 2: for each node of a GFT, all subtrees have disjoint semantics.

Before motivating both restrictions, we will first define the semantics of a GFT. As in the previous section, this semantics is a set of products. The definition of the previous section relied on the 1-1 correspondence between nodes and features, which does not exist here. Therefore, we first define a set of sets of nodes, instead of a set of products as in the previous section. This set of sets of nodes contains each set of nodes which satisfies

- It contains the root of the GFT.
- For each node in the set except the root, the set also contains its parent node.
- For each MandOpt node in the set, the set also contains all its mandatory child nodes.
- For each Or node in the set, the set also contains one or more of its child nodes.
- For each Xor node in the set, the set also contains exactly one of its child nodes.

The semantics of the GFT is now defined as the set of products which is obtained from this set of sets of nodes when each node is replaced by its feature. Where each feature tree is a GFT, it is seen that this definition coincides with the definition of the previous section when the GFT is a feature tree.

Although a GFT may contain multiple occurrences of a feature, we do not want multiple occurrences of features in products. This is the motivation of the first restriction above; it prevents multiple occurrences of features in products.

In a feature tree, different subtrees of a node do not contain equal features; this means that the semantics of these subtrees are disjoint. For a GFT, different subtrees of a node may contain equal features; however, we still want the semantics of these subtrees to be disjoint, as is expressed by the second restiction above. The reason for this is that computations might become inefficient otherwise. For instance, consider a GFT whose root node is an Xor node which has two subtrees and suppose these subtrees have N and M products respectively. When we know that these sets of products are disjoint we can conclude that the total number of products is N+M. Otherwise, we have to single out common products from both sets.

An important property of GFTs which is not valid for feature trees is that for each set of products there exists a GFT. Given a set of products, a corresponding GFT can be constructed as an Xor root node with

subtrees for each product; each subtree corresponds to a single product.

In the remainder of this section we present two algorithms, which deal with commitment to a feature and deletion of a feature of a GFT, respectively. These algorithms are generalisations of algorithms for feature trees which are given in [6].

The first algorithm computes, given a GFT T and a feature F, the GFT T(+F), whose products are precisely those products of T which contain F. The algorithm transforms T into T(+F) as follows:

1. If T does not contain F, T(+F) is NIL, else GOTO 2
2. If F is the feature of the root node of T, T(+F) is T, else GOTO 3
3. Execute 4, 5 or 6, depending on whether the root of T is a MandOpt node, an Or node or an Xor Node
4. Determine the unique subtree S which contains F, determine S(+F) recursively, and replace S by S(+F). If the root node of S was an optional node, make the root of S(+F) a mandatory node.
5. Determine the unique subtree S which contains F, determine S(+F) recursively, and replace T by a MandOpt node, with the same feature as T, which has S(+F) as mandatory subtree and all other subtrees of T as optional subtrees.
6. Determine the subtrees $S_1,..,S_n$ which contain F, determine $S_1(+F),..,S_n(+F)$ recursively, and replace $S_1,..,S_n$ by $S_1(+F),..,S_n(+F)$. Delete all other subtrees. If n=1, make the root node of T a MandOpt node, and its subtree a mandatory subtree.

The second algorithm computes, given a GFT T and a feature F, the GFT T(–F) whose products are precisely those products of T which do not contain F. The algorithm transforms T into T(–F) as follows:

1. If T does not contain F, T(–F) is T, else GOTO 2
2. If F is the feature of the root node of T, T(–F) is NIL, else GOTO 3
3. Execute 4, 5 or 6, depending on whether the root of T is a MandOpt node, an Or node or an Xor Node
4. Determine the unique subtree S which contains F and determine S(–F) recursively. If S is mandatory and S(–F) = NIL, then T(–F) is NIL. IF S is optional and S(–F) = NIL delete S from T. If S(–F) ≠ NIL then replace S by S(–F).
5. Determine the unique subtree S which contains F, determine S(–F) recursively. If S(–F) ≠ NIL, replace S by S(–F). If S(–F) = NIL, delete S. If T has only 1 subtree left, make its root node a MandOpt node, and its child a mandatory child.
6. Determine the subtrees $S_1,..,S_n$ which contain F and determine $S_1(–F),..,S_n(–F)$ recursively, and delete all other subtrees. For i=1,..,n, if $S_i(–F)$ = NIL, delete S_i, otherwise replace S_i by $S_i(–F)$. If T has no subtrees left, then T(–F) is NIL If T has only 1 subtree left, make its root node a MandOpt node, and its child a mandatory child.

In [6] we gave an implementation in Miranda of functions with type definitions

```
commit :: feature -> tree -> tree
delete :: feature -> tree -> tree
```

These functions, originally given for feature trees, need no modification to be applicable to GFTs as well. The function `commit` takes a feature F and a GFT T as arguments, and returns T(+F), as defined above. Likewise, the function `delete` returns T(–F).

4. From feature models to generalised feature trees

In [6] we showed how a feature model which consists of a feature tree and Requires and/or Excludes constraints can be transformed into a tree with the same semantics. Here we will generalise this method to general constraints, and show that the resulting tree is a GFT. Suppose we are given a feature tree T and a constraint C, which is a mapping from P, the set of all products with features of T to the Boolean values {True,False}. Find a partition of P such that C is constant on each part. For each part on which C is True, find a corresponding GFT, using the algorithms of the previous section. Finally obtain the GFT whose root node is an Xor node and which has the GFTs just found as child nodes. As an example, consider the constraint C to be "A requires B". Partition P into {P(+B), P(–A–B), P(+A–B). Here "+A" and "–A" denote restriction to products where A is present resp. absent. C is True on P(+B) and P(–A–B) and C is False on P(+A–B). GFTs for P(+B) and P(–A–B) are T(+B) and T(–A–B). The resulting GFT has an Xor root node and T(+B) and T(–A–B) as subtrees. Analogously, if C is "A excludes B", the new GFT has an Xor root node and T(–B) and T(–A+B) as subtrees.

The resulting trees are indeed GFTs. Restriction 1 is satisfied because all generated subtrees have the same Xor root node as parent node. Restriction 2 is satisfied because the semantics of the generated subtrees are the parts of a partition of P, and therefore have no common features.

In [6] we gave an implementation in Miranda of a function with type definition

```
elimConstr :: feature_model -> tree
```

The argument of this function is a feature model which consists of a feature tree and constraints of the forms "A requires B" and "A excludes B"; the function returns a corresponding GFT.

5. Analysis of feature models

In this section we show how the algorithms of the preceding section can be used to analyse feature models which consist of a feature tree and a number of constraints. In this implementation the constraints are restricted to be of the forms the function `elimConstr` can handle; these are the forms "A requires B" and "A excludes B",but other forms might be included as well, as described in the previous section.

Starting point of the analysis is a feature model consisting of the feature tree `f_tree` and the list of constraints `constraints`. The first step of the analysis is the computation of an equivalent GFT `gft`:

```
gft :: tree
gft = elimConstr (f_tree, constraints)
```

The function `elimConstr` here can be used if all constraints are of the forms "A requires B" and "A excludes B"; otherwise, the procedure described in the previous section should be followed.

In the remainder of this section we describe the computation of a number of properties of the specified software product line.

Existence of products

The feature model has products if and only if `gft` is not equal to NIL:

```
has_products :: bool
has_products = gft ~= NIL
```

Dead features

The dead features of the feature model are the features which occur in features but do not occur in `gft`:

```
dead_features :: [feature]
dead_features
   = features f_tree -- features gft
```

Here the function `features` computes a list of all features of a GFT:

```
features :: tree -> [feature]
features (MandOpt f ms os)
   = f : concat (map features (ms++os))
features (Or f fts)
   = f : concat (map features fts)
features (Xor f fts)
   = f : concat (map features fts)
```

Number of products

The number of products of the feature model is

```
nr_products :: num
nr_products = nrProds gft
```

where the function `nrProds` is given by

```
nrProds :: tree -> num
nrProds NIL = 0
nrProds (MandOpt nm ms os)
 = product (map nrProds ms) *
   product (map (+1) (map nrProds os))
nrProds (Xor nm fts)
 = sum (map nrProds fts)
nrProds (Or nm fts)
 = product (map(+1)(map nrProds fts)) - 1
```

List of all products

A list of all products of the feature model is

```
list_of_products :: [[feature]]
list_of_products = products gft
```

where the function `products` computes a list of products of a GFT:

```
products :: tree -> [[feature]]
products (MandOpt x ms os)
 = map(x:)(f(map products ms ++
             map([]:)(map products os)))
   where
   f [] =  [[]]
   f(xs:xss) = [u++v|u<-xs;v<-f xss]
products (Xor x fts)
 = map(x:)(foldl(++)[](map products fts))
products (Or x fts)
 = map(x:)(f(map products fts)--[[]])
   where
   f [] = [[]]
   f(xs:xss) = [u++v|u<-([]:xs);v<-f xss]
```

Products which contain a given set of features

A GFT whose products are precisely those products of `gft` which contain all features from a list `required_features` is:

```
gft2 :: tree
gft2 = gft_req_fts required_features gft
```

where the function `gft_req_fts` is defined by:

```
gft_req_fts :: [feature] -> tree -> tree
gft_req_fts [] t = t
gft_req_fts (f:fs) t
        = commit f (gft_req_fts fs t)
```

Minimal set of conflicting constraints

A set of constraints is in conflict with a feature tree if the feature model consisting of this tree and these constraints has no products, i.e when `gft` evaluates to NIL. A user, confronted with such a conflict, may want some explanation of this. A solution might be to provide the user with a smallest minimal set of constraints that conflict with the feature tree. A minimal set of constraints is a set which contains conflicting constraints, but has no proper subset whose constraints also conflict. A smallest minimal set of conflicting constraints can be computed by

```
confl_constr :: [constraint]
confl_constr = smsocc(f_tree,constraints)
```

where the function `smsocc` (smallest minimal set of conflicting constraints) is given by:

```
smsocc :: feature_model -> [constraint]
smsocc (t,[]) = []
smsocc (t,c:cs)
  = [c], if t2 = NIL
  = [], if set1 = []
  = c:set1, if set2 = [] \/ #set2>#set1
  = set2, otherwise
    where
    t2 = elimConstr (t,[c])
    set1 = smsocc (t2,cs)
    set2 = smsocc (t,cs)
```

This function, given the original feature model as argument, returns a list with a minimal set of conflicting constraints if `gft` equals NIL; otherwise it returns the empty list.

Explanation of dead feature

If `dead_features`, the list of dead features, is non-empty and contains the feature `dead_feature`, the user might want explanation why this feature is dead. As above, this explanation is a minimal set of constraints which causes the feature to be dead. It is given by

```
expl_dead_ft :: [constraint]
expl_dead_ft
  = explain (f_tree,constraints)
            dead_feature
```

where the function `explain` is given by

```
explain :: feature_model
           -> feature -> [constraint]
explain (t,cs) f
  = smsocc (t2,cs), if t2 ~= NIL
  = [], otherwise
    where
    t2 = commit f t
```

The arguments of this function are the original feature model and a feature from the list `dead_features`. It returns a minimal set of constraints which causes the feature to be dead. If the feature does not belong to `dead_features`, the empty list is returned.

6. Example

As an example, consider the feature tree T in Figure 1, which is adapted from [9].

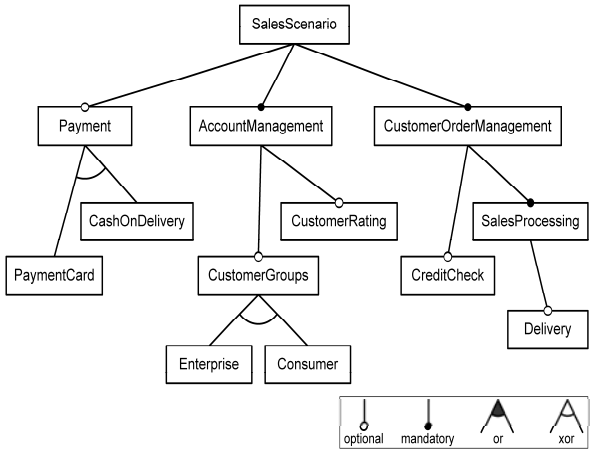


Figure 1. Example feature tree T

In Miranda, the definition of `f_tree` is

```
f_tree = MandOpt "SalesScenario"
         [n2,n3] [n1]
n1 = Xor "Payment" [n4,n5]
n2 = MandOpt "AccountManagement"
     [] [n6,n7]
n3 = MandOpt "CustomerOrderManagement"
     [n9] [n8]
n4 = MandOpt "PaymentCard" [] []
n5 = MandOpt "CashOnDelivery" [] []
n6 = Xor "CustomerGroups" [n10,n11]
n7 = MandOpt "CustomerRating" [] []
n8 = MandOpt "CreditCheck" [] []
n9 = MandOpt "SalesProcessing" [] [n12]
n10 = MandOpt "Enterprise" [] []
n11 = MandOpt "Consumer" [] []
n12 = MandOpt "Delivery" [] []
```

Our example feature model consists of this feature tree T and the 2 constraints: "CashOnDelivery excludes Consumer" and "Enterprise requires Consumer". So the list `constraints` is given by

```
constraints = [c1,c2]
c1 = Excludes "CashOnDelivery" "Consumer"
c2 = Requires "Enterprise" "Consumer"
```

The GFT `gft` is given in Figures 2, 3 and 4. It could have been computed with the function `elimConstr`, since both constraints are of the forms "A requires B" and "A excludes B"; however, we will illustrate the method to derive it which was given in section 4. If "Consumer" is present in a product, the constraints are satisfied iff "CashOnDelivery" is not present. If "Consumer" is absent in a product, the constraints are satisfied iff "Enterprise" is also absent. So the set of products P can be partitioned in such a way that the parts P(+Consumer–CashOnDelivery) and P(–Consumer–Enterprise) consist of the products which satisfy the constraints. Therefore, the equivalent GFT is given by a new Xor node which has T(+Consumer–CashOnDelivery) and T(–Consumer–Enterprise) as subtrees.

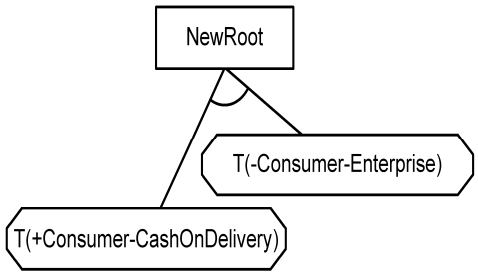


Figure 2. generalised feature tree `gft`, toplevel

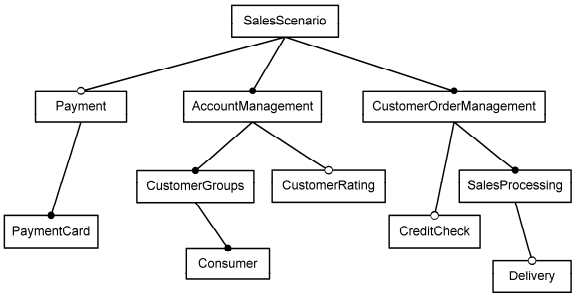


Figure 3. T(+Consumer–CashOnDelivery)

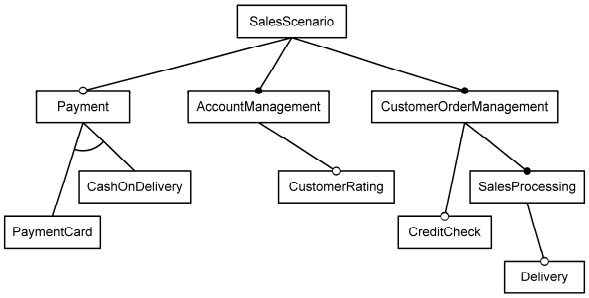


Figure 4. T(–Consumer–Enterprise)

The analysis of this example feature model proceeds as follows:

- `has_products` evaluates to `True`.
- `dead_features` evaluates to `["Enterprise"]`, showing that the feature "Enterprise"is dead.
- `nr_products` evaluates to `40`.
- `list_of_products` evaluates to a list of the 40 products (not shown for brevity).
- if `required_features` is defined as a list of features then `gft2` evaluates to a GFT which can be analyzed in the same manner.
- if `dead_feature` is defined to be `"Enterprise"` then `expl_dead_ft` evaluates to `[Requires "Enterprise" "Consumer"]` showing that the second constraint is on its own responsible for the deadness of `"Enterprise"`.

Now suppose that an extra constraint "SalesProcessing requires Enterprise" is added:

```
constraints = [c1,c2,c3]
c3 = Requires "SalesProcessing"
              "Enterprise"
```

Now `has_products` evaluates to `False` and `confl_constr` evaluates to `[Requires "Enterprise" "Consumer", Requires "Sales Processing" "Enterprise"]` which shows that the second and third constraints together form a smallest minimal set of constraints that conflict with the feature tree.

7. Computational Complexity

We have shown in [6] that the decision problem whether a feature model which is given by a feature tree and a set of constraints is NP-complete. Therefore, we cannot hope that the analysis of such a feature model can be performed in polynomial time in the worst case. Indeed, the construction of the GFT for the feature model takes a time which is exponential in the number

of constraints in the worst case. Also the algorithm for the computation of a minimal set of constraints which conflict with the feature tree and the algorithm which computes the minimal set of constraints which cause a feature to be dead are exponential in the number of constraints. However, once the GFT has been constructed, the algorithms for the existence of products, the number of products and the list of dead features are linear in the size of the GFT.

In the special case where the number of explicit constraints is 0, the intended GFT is the feature tree without modification. Then `has_products` belongs to O(1), and `nr_products` belongs to O(N). This certainly outperformes the other analysis methods mentioned in the introduction, as these methods require a transformation of the feature tree to another data structure; in the case of Binary Decision Diagrams this transformation requires even exponential time in the worst case. We expect that our method is more efficient than the other methods also in the case where the number of constraints is small. For instance, it has been shown in [6] that `nr_products` belongs to $O(N*2^M)$, where N is the number of features and M is the number of constraints. A more detailed comparison is planned as a future work.

8. Conclusion

We have introduced the concept of generalised feature trees and have shown how they can be used to analyse feature models which consist of a feature trees and additional constraints. Detailed algorithms have been given in the functional programming language Miranda. The algorithms are efficient when the constraints are a small number of "requires" and/or "excludes" constraints

9. Acknowledgments

This work is supported by the European Commission grant IST-33710 - Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE).

10. References

[1] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak and A.S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990).

[2] K. Czarnecki, U. Eisenecker, *Generative Programming: Methods Tools and Applications*, Addison-Wesley (2000).

[3] D. Benavides, P. Trinidad and A. Ruiz-Cortés, "Automated Reasoning on Feature Models", in: O. Pastor and J. Falcão e Cunha (Eds.): CAiSE 2005, Lecture Notes in Computer Science 3520, Springer-Verlag Berlin Heidelberg, 2005, pp. 491-503.

[4] D. Batory, "Feature Models, Grammars, and Propositional Formulas", in: H. Obbink and K. Pohl (eds.): Software Product Lines Conference 2005, Lecture Notes in Computer Science 3714, Springer-Verlag Berlin Heidelberg, pp. 7-20, 2005.

[5] K. Czarnecki and P. Kim, Cardinality-based Feature Modeling and Constraints: A Progress Report, in: Proceedings of the International Workshop on Software Factories, OOPSLA 2005, 2005.
http://softwarefactories.com/workshops/OOPSLA-2005/Papers/Czarnecki.pdf.

[6] P. van den Broek, I. Galvão and J. Noppen, "Elimination of Constraints from Feature Trees" in: Steffen Thiel and Klaus Pohl (Eds.), Proceedings of the 12th International Software Product Line Conference, Second Volume, Lero International Science Centre, University of Limerick, Ireland, ISBN 978-1-905952-06-9, 2008, pp. 227-232.

[7] P.-Y. Schobbens, P. Heymans, J.-Chr. Trigaux and Y. Bontemps, "Generic Semantics of Feature Diagrams", *Computer Networks* 51, 2007, pp. 456–479.

[8] D. Turner, Miranda: a non-strict functional language with polymorphic types, in: Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science Vol 201, J.-P. Jouannaud (ed.), Springer-Verlag, Berlin, Heidelberg, 1985, pp. 1-16.

[9] H. Morganho, J.P. Pimentão, R. Ribeiro, C. Pohl, A. Rummler, C. Schwanninger and L. Fiege, "Description of Feasible Industrial Case Studies", Deliverable 5.1, AMPLE Project, http://www.ample-project.net/ (2007).

Visualising Inter-Model Relationships in Software Product Lines

Ciarán Cawley, Steffen Thiel, Goetz Botterweck, Daren Nestor

Lero,University of Limerick
Limerick, Ireland
{ ciaran.cawley | steffen.thiel | goetz.botterweck | daren.nestor }@lero.ie

Abstract

Within Software Product Line (SPL) Engineering, Feature modelling is a prevalent mechanism for managing variability but is insufficient for describing it as a whole and for relating its different aspects. Other modelling techniques such as Decision modelling and Component modelling provide different views of the underlying SPL data. To facilitate certain approaches in product line engineering, such as tool-supported product derivation or automated analyses we need to describe the SPL with multiple inter-related models. In this paper, we discuss how to represent the relationships that exist between those models and present an approach for communicating the relationships using visualisation techniques. We also discuss the visualisation through example scenarios and argue its benefits.

1. Introduction

Software Product Line (SPL) Engineering focuses on identifying and managing the commonalties and variability of a set of software products such that core assets can be developed and (re)used to derive individual product variants with a minimum of cost [1]. Within industry, software product lines exist with thousands of variation points presenting a scale of variability where comprehension and manageability become difficult to achieve [2], [3]. The management of such a product line's variability is fundamentally key to its success. Particularly in the areas of modelling of the product line and product configuration, variability management can greatly impact the complexity that is involved when producing a new product from existing product line assets [4]. Visualisation is widely used in software engineering and has proven useful to amplify human cognition in data intensive applications. It takes abstract data and transforms it into a format that is presentable to humans. By leveraging the principles and techniques developed by visualisation communities [5] we can attempt to address the scale and complexity issues that challenge large scale software product line efficiency. Feature modelling is currently a prominent mechanism for managing software product line commonality and variability e.g. [6]. However, the authors will argue that a *feature* model in itself does not adequately describe a product line. Other mechanisms for managing the complexity of a product line, such as *decision* modelling [7] and *component* modelling [8] endeavour to present a different view of the underlying data that comprises a software product line. In this paper, we present a visualisation approach that attempts to address the challenges of representing multiple models and their relationships with each other.

In Section 2, we motivate the need for multiple models and why they need to be integrated. In Section 3 we discuss the challenges faced by this integration. In Section 4 we present our visualisation approach and in Section 5 summarise a tool framework that will incorporate it. Section 6 presents related work and Sections 7 and 8 outline future work and conclude the paper respectively.

2. Motivation

Most SPL research approaches focus on single development artefacts, represented by isolated models (e.g. [9] [10]). Commonly, these artefacts are features. While viewing a product line as a collection of features has many advantages, there are some disadvantages also. Some of the disadvantages include the problem of describing cross-cutting features, describing non-functional requirements, and the problems that arise in linking a feature to a concrete component (or set of components) that implement that feature.

There are numerous tasks that need to be performed by various stakeholders during the SPL engineering processes of domain engineering (development of core assets) and application engineering (specific product

variant derivation). Platform managers, domain engineers, product managers, application engineers, customers and developers all perform different roles in the process and require methodology and tool support that facilitates their specific tasks. In many of these cases, a feature model alone is either too detailed, or not detailed enough. Using separate models allows different facets of the product line to be managed in a focussed manner and supports stakeholder and task specific representation and manipulation. Section 3 discusses the resulting challenges that arise, namely the need to support the representation and affects of the relationships that exist once separate models are employed.

3. Inter-Model Relationships

The approach presented in this paper is based on the meta-models presented in [6]. These meta-models are not described here however the interested reader is directed to the relevant reference.

To exploit the benefits of a product line we need to connect the isolated models that describe the individual portions of the product line such as *decision* models, *feature* models and *component* models. In our approach, *decisions* provide a simplified high-level view onto *features* and can be used to abstract from details by asking a few major questions which are relevant for a particular stakeholder. A *component* model describes *components* that implement *features*. Making a *decision* can involve the selection of multiple *features*, each of which may require or exclude sets of other *features*. These *features* in turn may require or exclude sets of *components*. Furthermore, a relationship itself between two *features* may be implemented by a *component*.

Visualisation of the relationships within a feature model alone is challenging, and numerous approaches have been proposed, from filtered lists [7] to graph based views [11] to methods of only showing the relationships on demand [12]. With multiple models in place, visualising the relationships between each of them becomes even more difficult. Presentation and manipulation of the underlying data in the execution of specific tasks is impeded by the multilayered relationships that exist between them. For example as mentioned above, making a *decision* can involve the selection of multiple *features*, each of which may require or exclude sets of other *features*. These *features* in turn may require or exclude sets of *components*. In scenarios like this, stakeholders need to be presented with the relevant data using appropriate techniques that allow them to understand the current state and the impact of various changes that may be necessary. They also need to be able to make those changes easily. Crosscutting requirements are another major challenge. One way of describing a crosscutting requirement or feature is through use of *aspects*.

4. Relationship Visualisation Approach

This section presents an approach to visualising relationships that exist between the various models. The primary concept is to focus the attention of the visualisation on the *relationships* that exist between elements of one model and elements of other models (e.g. feature *implementedby* component). This is in contrast to traditional approaches which focus primarily on the *elements* (such as features or components) of the models. Such approaches subsequently describe how those elements are connected (such as node-link diagrams) [6], [11]. In this approach, the relationships themselves are the prominent visual element in the view. Using this concept, the visualisation aims to provide a high-level context of relationships relevant to given stakeholder tasks along with the ability to focus on specific data elements that support those tasks (examples are discussed later in this section). The ability of a stakeholder to interact, explore and query the visualisation is of primary importance. The following subsections explain and illustrate this approach providing example scenarios. Subsection 4.1 introduces the concept of a *relation point* while subsection 4.2 elaborates on its use within our approach.

4.1. Relation Points

Figure 1 illustrates a 3D space. Each axis in this 3D space represents a different model. *Features* are represented along the X-axis, *decisions* are represented along a vertical Z-axis and *components* along the Y-axis. A *relation point* is a point plotted in the 3D space where a *decision, feature* and *component* intersect, i.e. where a *decision* is *implemented by* a *feature* and a *feature* is *implemented by* a *component*. These two inter-model relationships connecting three separate models are represented by a single visual element.

Figure 1 serves to illustrate a *relation point* with a simplified example. In this example, the decision to include Hardware Platform B (vertical Z-axis) results in the spherical *relation point* being displayed. (For the purposes of this paper, it is suggested that *decisions* are made through an associated synchronised view such as a *decision* list.) Three functions are evident. Firstly, the existence of the *relation point* identifies that relationships are impacted within the system. Secondly, with simple interaction such as clicking on the *relation point*, the stakeholder is informed that the Software Upgradeability *feature* has been selected and

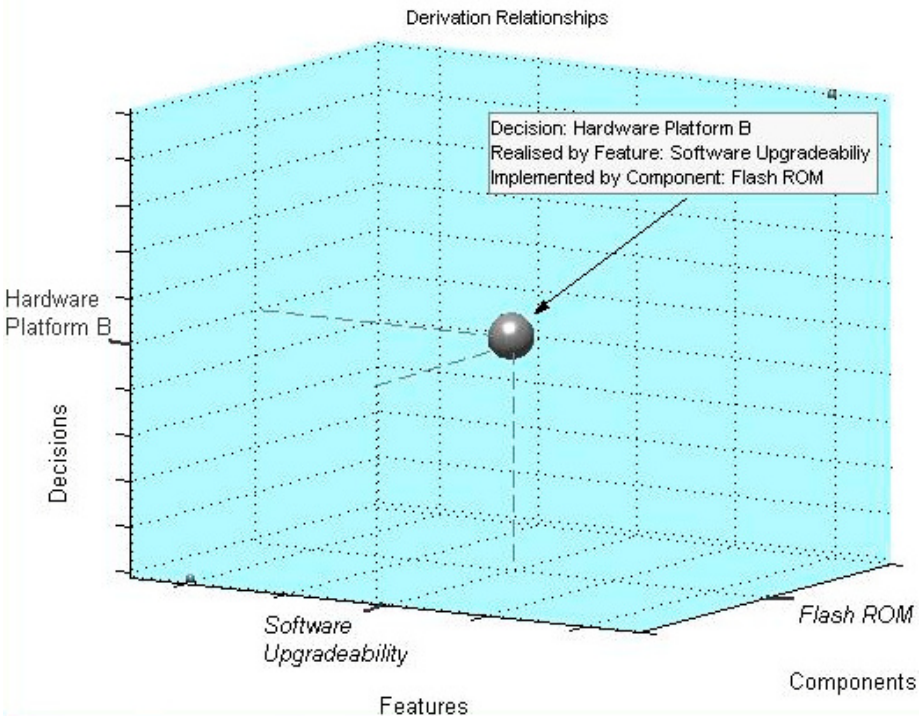


Figure 1. Relation Point Example

that it is *implemented by* the Flash Rom component. This is shown by displaying / highlighting the *decision*, *feature* and *component* names on each of the axes. Thirdly, through the use of layering (tooltip style functionality), the two relationships in this example, *decision* is *implemented by feature* and feature is *implemented by component*, are displayed textually providing more detail on demand.

This single visual element can also represent many more relationships and attributes without adding additional visual clutter. Visual techniques such as colour encoding the *relation point*, use of varying iconography to render the *relation point* and/or animation of the *relation point* can be applied to represent additional dependencies. For example, the use of a cube instead of a sphere to represent the *relation point* shown in Figure 1 could indicate the *feature* is related to a fourth or fifth model such as a business driver or quality model. By querying the *relation point* as above, the details of the relationship can again be provided on demand. Other such models can also be mapped to axes and swapped in and out replacing existing model to axis mappings. The mapping of a model to an axis is a simple linear sequential listing of the model elements.

It is important to remember that a *relation point* represents *relationships*. As a consequence, for example, if a *feature* is *implemented by* two *components* then two different *relation points* will be displayed. One will represent the relationship *feature implemented by component* 1 and the other will represent *feature implemented by component* 2 and both will represent the same relationship *decision implemented by feature.* How this type of visualisation can aid a stakeholder using interaction is discussed in the following subsection.

Intra-model relationships are also encapsulated such as a green *relation point* indicating that the specific *feature* associated with the *relation point* is required due to an intra-model *requires* relationship with another *feature*. As briefly mentioned earlier, with a substantial amount of additional information available, specific data can be acquired on demand through stakeholder interaction such as mouse movements and events. The next section elaborates more on some of these points through an example and discusses the role of interaction within the visualisation.

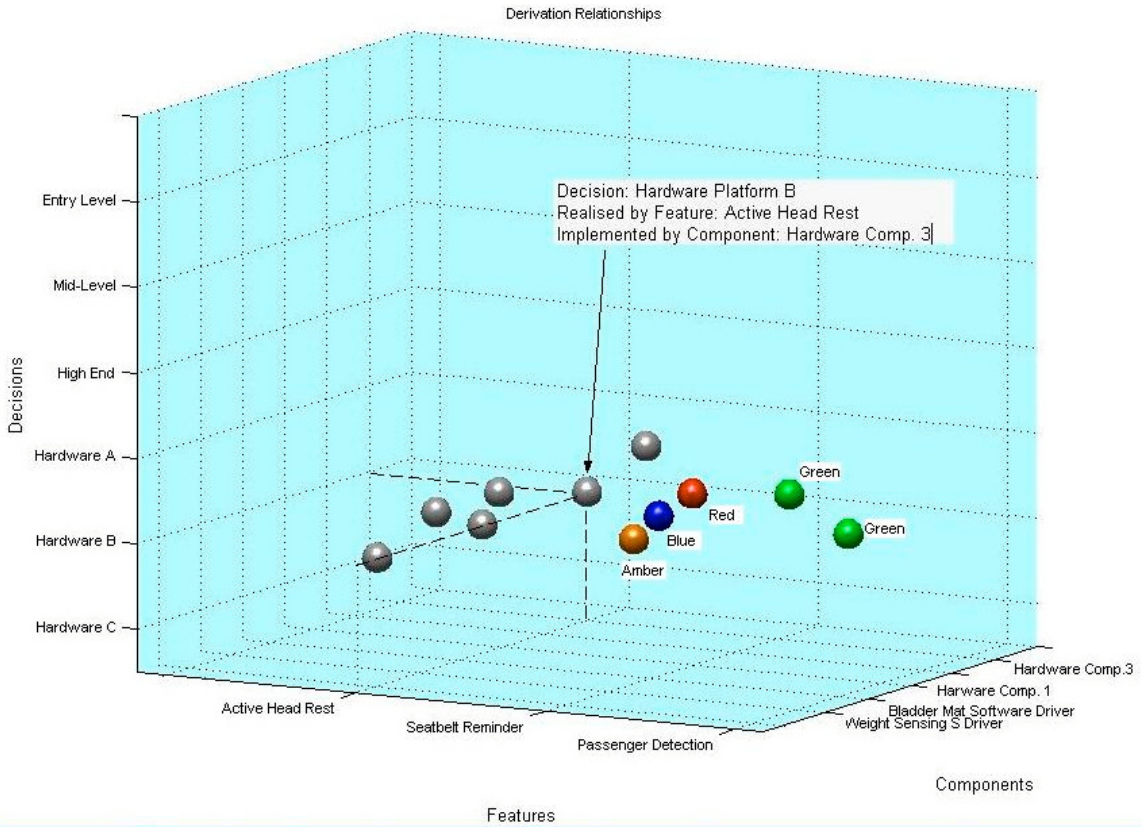


Figure 2. Interactive 3D Derivation Space

4.2. An Interactive Derivation Space

Following the example presented in Figure 1, Figure 2 presents a less simplified illustration of the visualisation. Here a stakeholder has resolved two *decisions*, to include hardware platform B and hardware platform C. On the resolution (making) of these *decisions*, eleven *relation points* are visualised. A grey *relation point* indicates a *decision implemented by feature* relationship and a *feature implemented by component* relationship. With 6 grey *relation points*, 12 relationships in total are visualised. If a *relation point* is green it identifies an additional relationship which represents *feature requires feature*. If a *relation point* is red it identifies the additional relationship *feature excludes feature*. If a *relation point* is blue it identifies the additional relationship *feature recommends feature*. If a *relation point* is amber it identifies the additional relationship *feature problematic* with *feature*. In total, 27 relationships incorporating 3 separate models are visualised.

4.2.1. Stakeholder Interaction

As with any visualisation where a stakeholder is required to perform specific tasks, appropriate interaction is needed. We argue that this 3D visualisation allows for a substantial number of relation points to be rendered without causing information overload. This allows for a large amount of relational information to be presented. Further work is required to identify a balance between the number of *relation points* displayed based on specific tasks and the onset of such information overload.

3D visualisations afford the use of the world-in-hand metaphor where the stakeholder can rotate it as a whole in any direction. This kind of interaction allows a familiar exploration of the presented data where the stakeholder can manipulate the visualisation to gain an optimum view dependent on their interest. For example, where aspect oriented programming is used and in the event of say 50 relation points representing say 150 relationships, an application engineer can orient the visualisation so that the best possible view of all *aspect* implemented *features* is shown. Distortion techniques such as blurring and/or transparency could be applied by the application engineer to distort all non aspect implemented *features* thus highlighting what is of interest without losing the context of the rest of the relationships impacted by the *decisions* made.

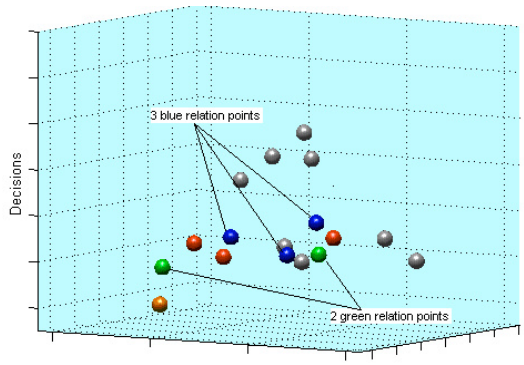


Figure 3. High Impact Decision Example

As discussed in 4.1, *relation points* themselves can be interacted with through mouse movements and clicking. In this way details-on-demand can be rendered when required.

4.2.2. View Interaction

As was briefly discussed earlier, synchronising this visualisation with other views provides a number of additional functions. Providing a simple *decision* view such as a list allows a stakeholder to abstract details of the system at a high level. The approach presented in this paper can then act as an automatic rendering of the size and nature of the impact on the system as a whole providing immediate visual feedback through colour, shapes and iconography. Other views such as a traditional tree diagram [6] which provide contextual information for individual models can also be synchronised with our approach. For example, by clicking on a *relation point*, the nodes and links in a separate, associated tree view representing the relevant relationships between the different models can be highlighted. This affords a different perspective that may be more suited and functional for different stakeholder tasks.

4.3. Task Support

The primary aim of this visualisation is to provide cognitive support to various stakeholders which aids *feature* configuration during product derivation. The rest of this subsection discusses this in the context of the three models in the example.

When a stakeholder resolves a *decision*, this approach provides immediate visual feedback regarding the size and nature of the impact of that *decision* on the rest of the system. It does this by specifically identifying what *features* and what *components* are affected and by identifying the type of relationships that govern their configuration. For example, if a large number of *relation points* appear in the visualisation then the stakeholder immediately comprehends that the *decision* has a significant impact. Figure 4 shows a conceptual illustration of how such a scenario might be visualised. Using the colour encoding scheme described earlier, we argue that it is easily ascertained that out of the 17 *relation points* displayed, 8 grey *relation points* represent *features* directly related to the *decision* that would automatically be selected. 2 green *relation points* represent *features* that would also automatically be selected due to the selection of other *features*. 3 red *relation points* represent *features* that would be automatically excluded from the configuration. 3 blue *relation points* represent recommended *features* that require attention for possible selection and one amber *relation point* represents a problematic *feature*. In total, 43 relationships incorporating 3 different models are visualised.

If a large number of red *relation points* exist then a large number of *features* are being excluded and perhaps further investigation of the excluded *features* is advised. Furthermore, if a large number of amber *relation points* exist then the stakeholder understands that there could possibly be substantial issues associated with this configuration and further investigation is imperative. High risk and high cost *features* can be treated similarly. Figure 3 shows another conceptual illustration presenting how a high *feature* exclusion view might be visualised. Here, 6 red *relation points* represent 6 *features* excluded due to other included *features* directly related to the *decision*.

This visualisation provides a high-level context of all relationships that are impacted by a *decision* choice. This context can be rotated, zoomed and panned to facilitate exploration of the data and to provide optimum views. This gives the stakeholder an overview perspective allowing them to gauge the current system impact as a whole. By applying distortion techniques, as described earlier, specific

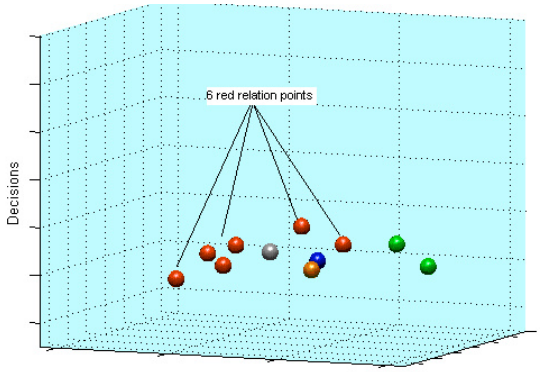


Figure 4. High Exclusion Example

aspects of the visualisation can be highlighted and explored without losing the context. For example, use of blurring and transparency to lowlight all *relation points* that do not represent *feature problematic* with *feature* relationships. This has the effect of highlighting all problematic *features* where possible issues could arise.

As was discussed in section 4.2, this context can be used as an anchor to drive other views. By clicking on a relation point, alternative views can display more specific information using different visual metaphors.

With the above context, encoding, interaction and exploration techniques in place, the investigation of specific details is afforded. For example, a stakeholder, having highlighted all problematic relationships can click on those relation points and view more specific information such as what feature it is problematic with, providing an issue discovery mechanism.

5. Framework

We are currently developing a tool framework which provides an infrastructure whereby separate models and their relationships can be instantiated and used as the back-end for various visualisations that support a variety of tasks. The details of this framework are out of scope for the current paper but a summary is provided here to relate the approach presented above.

Figure 5 illustrates the outline of the framework. Meta-models describing the SPL data and the relationships that exist allow separate models to be created. The illustration shows three such models, *decision*, *feature* and *component* as well as the existence of the inter-model relationships. These models and their defined relationships represent the back-end data that is used to drive different front-end visualisations that will support product derivation tasks. The approach presented in this paper ("3D Relation View" in Figure 5) will be incorporated as one such visualisation by implementing it using the java3D API. Other possible visualisation examples are shown in Figure 5 such as a simple List View for displaying a list of *decisions* or a Tree View showing a *feature* hierarchy and the related *component* implementations.

The framework uses the Eclipse Modelling Framework (EMF) and a java implementation provides a concrete platform to support the front-end visualisations. It is envisaged that this framework will be developed further to support additional models, relationships and visualisations that will aid product derivation in software product lines.

6. Related Work

Kumbang [13], pure::variants [14], FeaturePlugin [15] and Gears [16] are examples of tools that aim to support product derivation in SPL engineering. All of these approaches primarily employ traditional visualisations to manage the data representation and manipulation of various models and relationships. Foremost of these is the use of simple lists and hierarchical tree views similar to those traditionally

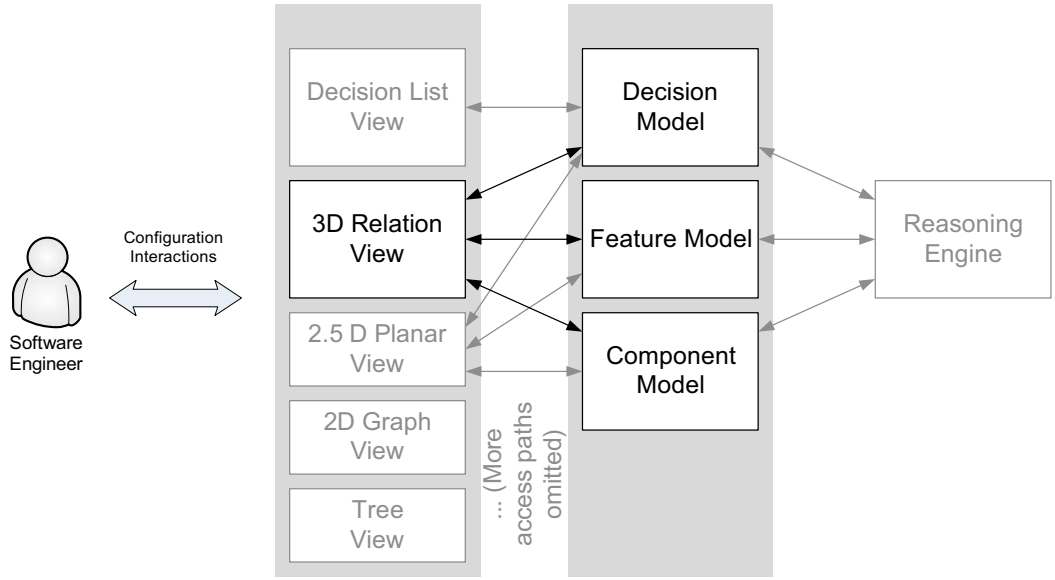


Figure 5. Framework Outline

used for displaying file systems. Such visualisations, though familiar, lack evidence of their effectiveness with large scale product lines. Our approach addresses the tasks of data representation and manipulation from a non-traditional perspective. A relationship-centred visualisation is used in an attempt to manage the complexity and scalability challenges that arise with large software product lines.

The DOPLER [7] tool also supports product derivation and while traditional lists and hierarchical tree views are primarily employed, there exists a mechanism whereby other visualisations can be incorporated. These visualisations can implement graph layout algorithms which again primarily focus on traditional node-link diagrams. While the focus of the tool is on *decisions* and *assets*, there is no support for additional models and their inter-relationships.

Unlike any of the tools listed above or others such as XFeature [17], V-Visualize [11] or FeatureMapper [18], our approach attempts to harness the richness afforded by 3D visualisation to address complexity and scalability issues.

Other software visualisation tools such as VISMOOS [19] and MUDRIK [20] provide interesting uses of 3D implementations to support cognition. However these tools and other 3D work such as that by Balzer et al [21] do not support SPL product configuration and are primarily aimed at understanding and not process support.

Other work by Robertson et al [22] and Risden et al [23] are examples of 3D information visualisations where some evaluation of their effectiveness compared with 2D equivalents has been performed. Both papers suggest that in some situations there was no perceived benefit in having a 3D visualisation while in others there was a marked increase in task performance efficiency. This work serves to provide some evidence that 3D techniques can be effective in certain circumstances.

7. Future Work

It is planned to perform further literature review where evaluations of 3D visualisations and/or comparisons of the effectiveness of 3D versus 2D visualisations have been performed.

A specification formalising how this relationship visualisation concept will be implemented is intended. As discussed in this paper, a 3D visualisation environment provides the primary context within which this concept will be exploited. An implementation using the java3D API will be developed and incorporated into the tool framework discussed earlier.

With this visualisation and framework in place, initial user tests are planned to ascertain the effectiveness of this approach to support cognition during product derivation.

8. Conclusion

This paper motivates the need for multiple models in Software Product Line engineering to support management of the large data sets that can exist. It further discusses the requirement and challenges of integrating those separate models to aid different stakeholders in various tasks. A primary challenge is managing the complexity and scale of inter-model (and also intra-model) relationships so that their representation and manipulation is not overwhelming for the stakeholder.

An approach to managing this complexity in relation to product derivation is presented which employs visualisation techniques for the representation and manipulation of the underlying data. This approach focuses on representing the relationships that exist between different models as the primary visual element in the view. A 3D visual environment is employed to render these relationships to support stakeholder cognition during product derivation.

A tool framework that provides an infrastructure within which this visualisation and others can be incorporated is outlined. This framework provides the necessary back end data structures and models to support the visualisation front-end.

It is argued through example scenarios that this visualisation can benefit different stakeholders during product derivation by providing an overview context while allowing specific information to be presented and manipulated without loss of that context.

9. Acknowledgements

This work is partially supported by Science Foundation Ireland under grant number 03/CE2/I303-1.

10. References

[1] K. Pohl, G. Böckle, and F. v. d. Linden, *Software Product Line Engineering: Foundations, Princi-ples, and Techniques*, 1st ed. ed. New York: Springer, 2005.

[2] S. Deelstra, M. Sinnema, and J. Bosch, "Product Derivation in Software Product Families: A Case Study," *Journal of Systems and Software,* vol. 74, pp. 173-194, 2005.

[3] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber, "Introducing PLA at Bosch Gasoline Systems: Experiences and Practices," in *SPLC 2004*, Boston, MA, USA, 2004, pp. 34-50.

[4] M. Sinnema and S. Deelstra, "Classifying variability modeling techniques," *Information and Software Technology,* vol. 49, pp. 717-739, 2007.
[5] C. Ware, "Information Visualisation: Perception for Design," in *Morgan Kaufmann Series in Interactive Tecnology*, 2nd ed: Morgan Kaufmann, 2004.
[6] G. Botterweck, S. Thiel, D. Nestor, S. B. Abid, and C. Cawley, "Visual Tool Support for Configuring and Understanding Software Product Lines," in *The 12th International Software Product Line Conference (SPLC08)* Limerick, Ireland, 2008.
[7] R. Rabiser, D. Dhungana, and P. Grünbacher, "Tool Support for Product Derivation in Large-Scale Product Lines: A Wizard-based Approach," in *1st International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2007)* Tokyo, Japan, 2007.
[8] R. v. Ommering, F. v. d. Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software," *IEEE Computer,* vol. 33, pp. 78-85, 2000.
[9] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21," Software Engineering Institute, Carnegie Mellon University 1990.
[10] K. Czarnecki and S. H. a. U. Eisenecker, "Staged configuration using feature models," in *Proceedings of the Third Software Product Line Conference* Boston MA: Springer, 2004.
[11] D. Sellier and M. Mannion, "Visualizing Product Line Requirement Selection Decisions," in *1st International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2007)* Tokyo, Japan, 2007.
[12] C. Cawley, D. Nestor, A. Preußner, G. Botterweck, and S. Thiel, "Interactive Visualisation to Support Product Configuration in Software Product Lines," in *Proceed-ings of 2nd International Workshop on Variability Modeling of Software-Intensive Systems (VAMOS 2008)* Essen, Germany, 2008.
[13] T. Asikainen, T. Männistö, and T. Soininen, "Kumbang: A domain ontology for modelling variability in software product families," *Advanced Engineering Informatics,* vol. 21, pp. 23-40, 2007.
[14] pure-systems GmbH, "Variant Management with pure::variants," http://www.pure-systems.com, Technical White Paper, 2003-2004.
[15] M. Antkiewicz and K. Czarnecki, "FeaturePlugin: Feature Modeling plug-in for Eclipse," in *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, Vancouver, BC, Canada, 2004, pp. 67--72.
[16] Biglever Software, "Gears," http://www.biglever.com.
[17] O. Rohlik and A. Pasetti, "XFeature," http://www.pnp-software.com/XFeature/Home.html.
[18] F. Heidenreich, I. Savga, and C. Wende, "On Controlled Visualisations in Software Product Line Engineering," in *2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2008)* Limerick, Ireland, 2008.
[19] O. Rohr, "VisMOOS (Visualization Methods for Object Oriented Software Systems)," University of Dortmund, http://ls10-www.cs.uni-dortmund.de/vise3d/prototypes.html, 2004.
[20] J. Ali, "Cognitive support through visualization and focus specification for understanding large class libraries," *Journal of Visual Language and Computing,* 2008.
[21] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz, "Software Landscapes: Visualizing the Structure of Large Software Systems," in *Symposium on Visualization (VisSym 2004)* Konstanz, Germany: Eurographics Association, 2004.
[22] G. Robertson, K. Cameron, M. Czerwinski, and D. Robbins, "Polyarchy Visualization: Visualizing Multiple Intersecting Hierarchies," in *Conference on Human Factors in Computing Systems* Minneapolis, Minnesota, USA.: ACM, 2002.
[23] K. Risden, M. P. Czerwinski, T. Munzner, and D. B. Cook, "An initial examination of ease of use for 2D and 3D information visualizations of web content," *Int. J. Human-Computer Studies,* pp. 695-714, 2000.

Modeling Variation in Production Planning Artifacts

Gary J. Chastek
Software Engineering Institute
gjc@sei.cmu.edu

John D. McGregor
Clemson University
johnmc@cs.clemson.edu

Abstract

Production planning and variation modeling are interdependent parallel activities critical to the success of a software product line. Software product line organizations design a production capability to satisfy their business goals. That production capability is dependent on and must support the full range of product variation. Current techniques for variation modeling identify and handle variations among products but fail to recognize variations that result from business goals such as rapid time to market which are satisfied by how products are built. In this paper we present a view of our production planning technique and describe our preliminary research into its relation to variation modeling.

1. Introduction

Variation distinguishes a software product line from single product development. A variation is any concern that will vary from one product to another or that may vary in time. Product feature models alone are inadequate for modeling the full range of variations the software product line organization must manage. They capture product but not production variations. Goals like mass customization are production rather than product issues, and can yield strategically-significant variations. The variations in how products are built are a direct result of the goals of the organization. For example, a business goal to compete in the global market can lead to variations in testing processes used depending upon the market for which each product is intended.

These variations are the result of strategic decisions made during early product line planning activities of the "What to Build" (WTB) pattern [1], shown in Figure 1. Scoping uses the marketing analysis to identify the members of the product line. During this activity variations in features among the products are identified. The market analysis identifies market segments and analyzes the differences among them. Building the business case requires the creation of a justification for the use of the organization's assets to create and operate the product line organization. It also provides the opportunity to identify additional variations that are not directly related to product content but that are related to product production. We will call the variations found during the WTB activities strategic variations. A change in a strategic variation results in a change to at least one of the WTB artifacts, and vice versa. For example, expanding the scope of the product line may result in additional test requirements if the additional product is intended for a new market.

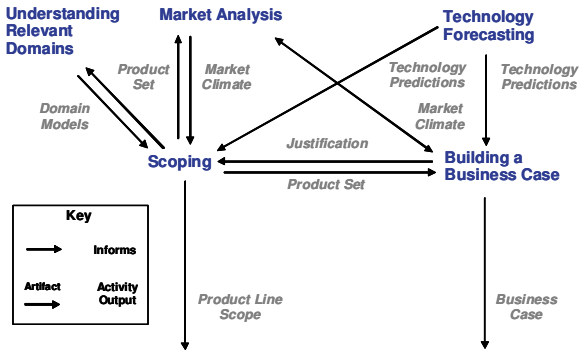


Figure 1 What to build pattern

In addition to strategic we also define tactical variations as those variations that arise from the resolution of strategic variations. For example, the development process will have a step in test planning where the levels of test coverage are determined by the intended market. Tactical variations typically do not necessitate a change in a strategic variation but could if, for example, the choice an architectural mechanism provided the possibility of supporting a wider range of products.

The progression from strategic to tactical variation and ultimately to variation point corresponds to positions along the Variation axis in the space defined in [2], as illustrated in Figure 2. The strategic variations

represent "what concerns" are important and are placed further out on the Variation axis than the tactical variations that represent "how" those concerns are addressed. Similarly, tactical variations are placed further out than variation points since tactical variations represent the broad concern while a specific variation point addresses only a portion of that concern. Finally, a bound variation point represents a fixed point in the two-dimensional single-product plane. This progression corresponds to the hierarchical separation of concerns described by [3].

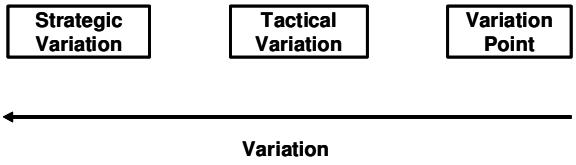


Figure 2 Variations in context

The goal of this paper is to describe our preliminary research into the relation between variation modeling and production planning. Its contribution is to identify key software product line variations not currently considered (i.e., variations related to how products are produced[1]) and to extend the work of Berg et al [2] by explicitly identifying strategic and tactical variation points.

The subsequent sections describe our work in production planning, variations in the production system and their relation to variation modeling in general, and finally the current directions of our ongoing research.

2. Production Planning Artifacts

How a software product line organization builds its products is a system, the production system [5], that has both functionality (e.g., the development tools employed) and quality attributes (e.g., how quickly a specified product can be delivered). Production planning devises a production system that

- Satisfies the organization's goals and constraints for its product line
- Coordinates the design of the core assets with the production system
- Communicates the effective use of the production system to the product developers

This is achieved by formulating a production strategy [6], constructing a production method [7], and documenting the production process in a production plan [8]. The production strategy links the business goals of the software product line organization to its means of product production. Porter's Five Forces model [9] is used to derive strategic actions that the production system must provide. For example, the organization can resolve the force of potential entrants (i.e., new competitors) into the market by taking the strategic action of automating product production and thereby reducing the per-product costs.

The production method defines an overall implementation approach that coordinates the efforts of the core asset and product developers. Method engineering techniques are used to define a method specifically for the needs of the product line organization. The production method contains processes, tools, and models that are used to implement the strategic actions related to product production. Efficiencies in production can be achieved by the elimination of inconsistencies across the processes, models, and tools. For example, the automating product production strategy can be partially implemented by using automated test generation tools.

The production plan communicates the production process details to the product developers. A generic production plan is developed as a core asset. Each core asset has "an attached process that specifies how it will be used in the development of actual products." [1] The plan is made product-specific by adding in the attached processes from the core assets selected for use by the product builders. For example, the production plan would specify the parameters to be used in instantiating the test generation facility.

The representation of variation points in the production planning artifacts varies by organization but should be compatible with the approaches being used in the organization's core assets. Ultimately the variation point representation must support the production of products in accordance with the goals for the software product line.

3. Production Planning Process and Modeling Variations

Each of the artifacts produced during production planning provides a different view on the variation model. During the early product line planning activities, described in the WTB pattern, variations are identified based on the differences among product feature sets and the differences in production techniques as identified from the business goals and

[1] Schmid and Eichelberger have addressed binding time as a meta-variability using aspects [4].

market analysis. These strategic variations are a primary input into the production planning process. As production planning moves from strategy development to method development to plan construction, the view of production becomes more concrete and focused.

3.1 Production Strategy

The production strategy defines the overall approach to producing products and begins the production planning process that ultimately results in the resolution of variation points. The strategic variations, identified during the analysis of the artifacts that result from applying the WTB pattern, lead to the identification of tactical production issues. For example, the strategic variation of addressing the delivery needs of a diverse market might translate into a tactical variation in the production process, e.g. waterfall vs. agile, that would be selected by the product builders for a specific product.

Table 1 Porter's forces

Porter Forces	Strategic Actions	Example impacts on variation
Potential Entrants	Leverage economies of scale	Minimize variant choices to get maximum use from each variant
Substitutes	Raise the cost of switching to another product	Provide some minimal variant implementations for each variation to allow for a low-cost product to attract or retain customers
Suppliers	Commoditize required components	Implement variations behind standard interfaces to maximize the number of potential suppliers
Buyers	Differentiate from other products	Maximize the number of variations to provide flexibility
Competitors	Improve features	Expand the number of variants available per variation to add features for customers

During production strategy development, strategic actions are identified as a means of resolving each of the forces identified using Porter's Five Forces strategy development model. The strategic actions that resolve one force can be at odds with another action resolving a different force leading to tradeoffs between those actions. Table 1 shows examples of strategic actions and corresponding impacts on tactical variations.

For example, the Potential Entrants force can be resolved by the strategic action of leveraging economies of scale, implying a minimization of the number of variant choices. Similarly the Buyers force can be resolved by differentiation from other products, leading to the maximization of the number of variations. This conflict can be resolved by maximizing the number of variations but minimizing the number of variants for each variation. This provides flexibility without a large upfront investment with the variation point providing the option to later expand the number of variants available.

3.2 Production Method

Method engineering designs constructs and adapts processes, models and tools for the development of software systems [10]. Method fragments are coherent pieces of development methods. The method engineer considers the characteristics of the organization and the current project to define a development method that will meet the specific needs of the project. In a software product line organization this includes the business goals and the variations in how the products will be produced.

The production method specifies how products will be built and directly affects how core assets are designed to support product building. The decisions about how to resolve the tactical variations are part of the method engineering activity that defines the production method. The production method uses a model of the tactical production variations to identify points of variation in the assets, see Figure 3.

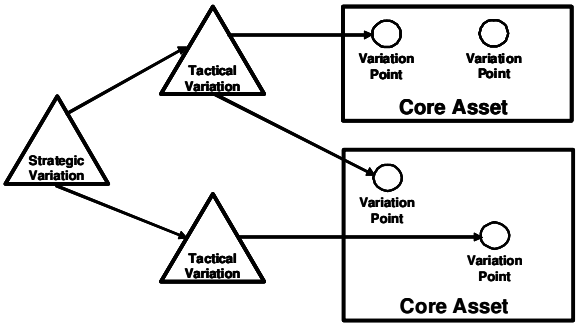


Figure 3 Variations to variation points

The model consists of the decisions that must be made regarding which portions of the production method to use in a specific situation. For example, if multiple deployment platforms are targeted, the

product builder would need to switch between different memory models and the tools specific to each. The production method would capture this variation in a process description and would specify the tools and models for both platforms.

The method engineers responsible for creating the product method determine the set of variation mechanisms from which the asset developers will be allowed to select for implementing their products. For example, aspect-oriented programming may be identified as a useful technique to include in the production method. Core asset developers will be able to create an asset that includes a basic unit and a set of aspects from which the product builder may select. The attached process for that asset would provide instructions for using the aspect weaver and other related tools and for incorporating the actions related to aspects into the production method.

The production strategy is a high-level answer to the question: "How can product development satisfy the organization's goals for the software product line?" The strategy provides a direct link between the product line goals and the means of product production. For example, a product line goal of "faster time to market" could lead to a strategy in which automation is used wherever possible.

3.3 Production Plan

The production plan guides product builders through building their specific product using the core assets. There are two "flavors" of the production plan: the generic and the product-specific.

The generic production plan is largely constructed from method fragments which correspond to the production variations. The method fragments are coordinated with the attached processes of specific core assets. The generic production plan, which is a core asset, is incomplete because it includes a model of the product variations. The variation points in this model represent decisions to be made by the product builder.

The generic production plan is instantiated into the product-specific production plan using the information in the attached processes of the core assets selected to satisfy the product variations, see Figure 4.

When a specific core asset is selected for use in building a product, its attached process is added to the emerging product-specific production plan. That includes adding a method fragment to the production method for that product just as the information provided by the core asset is added to the product. The generic production plan provides the context for making the tactical decisions required to select a variant from among those possible at each variation point. The fully instantiated product-specific production plan has no unbound variation points, but the product may still have unbound variation points.

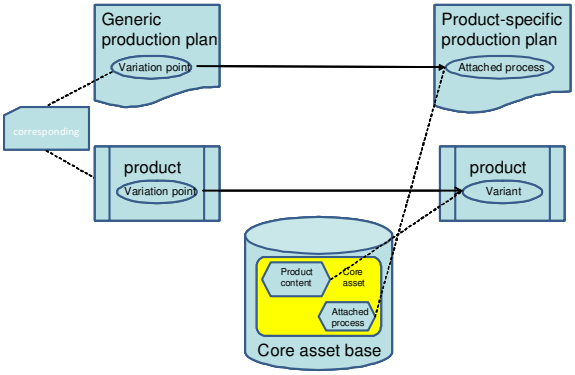


Figure 4 Production plan instantiation

3.4 Example

Bundling is a business strategy that groups a set of the products in the product line for sale as a unit. The products vary from each other but share a common high level purpose. A suite of word-processing, presentation graphics, and spreadsheet products are bundled based on a user's role in an organization. A security package bundles several products that all relate to the same general purpose - ensuring the security of a computer. The distribution of variants for a single feature across the products in the unit is a strategic variation. Some specific strategic variations include the range of operating systems to be supported or the variety of human languages to be supported.

The strategic variation of having different functionality based on a common purpose leads to a number of tactical variations. For example, file formats vary because different users handle very different types of information. The most efficient storage format for one type of information is different for other types of information. Another tactical variation is the variation among user interfaces. Some of the interfaces will be graphical while others will be text-oriented.

The tactical variations occur at places in core assets where product-specific information is required. The tactical variation in user interfaces results in a number of variation points. One such point might address the look and feel. Many windowing systems provide the ability to select different skins, menu styles, and window borders.

These variation points often correspond to variation points in the production method. For example, different products can require different binding times which in turn can require different design and construction techniques. Choosing one windowing systems could result in binding variation decisions at installation time while another windowing system would support configuration changes at any time.

4. Conclusions and Future Work

Variation modeling and production planning need to be cooperating activities. Production planning does not resolve product variations but they are vital to its success. At the very least the production method must provide the means for resolving all product variations. Frequently, product variations lead to variations in both the production method and production plan.

Variation in a software product line organization includes differences in the feature sets of products and in the system used to produce the products. These differences are captured in the assets related to product production. The production system must support these variations. That involves mapping the strategic variations into the core asset specifications including those of the production system itself.

The goals for a software product line impose goals and constraints on its production system. The production strategy, production method, and production plan provide a natural progression from strategic to tactical to concrete instructions for the product builder.

A number of issues require further investigation. Our techniques for identifying production variations are still intuitive and need more complete definition. The relationships between product and production variations and between strategic and tactical variations are not fully clarified. For example, it seems that each specific variation point has some element of both product and production associated with it. Is this true only for code-based assets or for assets in general?

There are multiple method engineering techniques that provide support for reusable pieces, i.e., method fragments. Which of these techniques can be used during method engineering to ensure that the fragments resulting from the different variants can be composed into a useful method? Will those techniques scale given the strategic levels of reuse encountered in a software product line organization?

Both production planning and variation management are central to product line success. The techniques presented in this paper illustrate that these two important ideas should work together to enhance product line operation.

5. Acknowledgments

Special permission to reproduce the Paper "Modeling Variation in Production Planning Artifacts" by Gary Chastek and John McGregor, Copyright 2008 by Carnegie Mellon University, is granted by the Software Engineering Institute.

6. References

[1] Clements, P. and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison Wesley, Reading, MA (2002).

[2] Berg, K.; Bishop, J.; & Muthig, D.: Tracing Software Product Line Variability – From Problem to Solution Space, Proceedings of SAICSIT 2005, pp 182 –191.

[3] Tarr, P.; Ossher, H.; Harrison, W.; Sutton, S. M.: N Degrees of Separation: Multi-dimensional Separation of Concerns. ICSE'99, 1999, pp. 107 – 119.

[4] Schmid, K. and Eichelberger, H.: Model-Based Implementation of Meta-Variability Constructs: A Case Study using Aspects, http://www.vamos-workshop.net/2008/papers/VAMOS08_07.pdf

[5] Chastek, G.; Donohoe, P.; McGregor, J. D.: A Production System for Software Product Lines, SPLC 2007, pp. 117-128, 11th International Software Product Line Conference (SPLC 2007), 2007.

[6] Chastek, G.; Donohoe, P.; McGregor, J. D.: Formulation of a Production Strategy for a Software Product Line. Technical Note, CMU/SEI-2008-TN-023, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 2008.

[7] Chastek, G.; Donohoe, P.; McGregor, J. D.: Applying Goal-Driven Method Engineering to Product Production in a Software Product Line. Technical Report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, to appear.

[8] Chastek, G.; McGregor, J. D.: Guidelines for Developing a Product Line Production Plan, Technical Report, CMU/SEI-2002-TR-006, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, June 2002.

[9] Porter, M. E., *Competitive Strategy*, Free Press, 2004.

[10] Brinkkemper, S. Method Engineering: Engineering of Information Systems Development Methods and Tools, Information and Software Technology 38 (1996) 275-280.

A Formal Semantics for Multi-level Staged Configuration

Andreas Classen,* Arnaud Hubaux and Patrick Heymans
PReCISE Research Centre,
Faculty of Computer Science,
University of Namur
5000 Namur, Belgium

E-mail: {acs,ahu,phe}@info.fundp.ac.be

Abstract

Multi-level staged configuration (MLSC) of feature diagrams has been proposed as a means to facilitate configuration in software product line engineering. Based on the observation that configuration often is a lengthy undertaking with many participants, MLSC splits it up into different levels that can be assigned to different stakeholders. This makes configuration more scalable to realistic environments. Although its supporting language (cardinality based feature diagrams) received various formal semantics, the MLSC process never received one. Nonetheless, a formal semantics is the primary indicator for precision and unambiguity and an important prerequisite for reliable tool-support.

We present a semantics for MLSC that builds on our earlier work on formal feature model semantics to which it adds the concepts of level *and* configuration path*. With the formal semantics, we were able to make the original definition more precise and to reveal some of its subtleties and incompletenesses. We also discovered some important properties that an MLSC process should possess and a configuration tool should guarantee. Our contribution is primarily of a fundamental nature, clarifying central, yet ambiguous, concepts and properties related to MLSC. Thereby, we intend to pave the way for safer, more efficient and more comprehensive automation of configuration tasks.*

1 Introduction

Feature Diagrams (FDs) are a common means to represent, and reason about, variability during Software Product Line (SPL) Engineering (SPLE) [10]. In this context, they have proved to be useful for a variety of tasks such as project scoping, requirements engineering and product configuration, and in a number of application domains such as telecoms, automotive and home automation systems [10].

The core purpose of an FD is to define concisely the set of legal *configurations* – generally called *products* – of some (usually software) artefact. An example FD is shown in Figure 1. Basically, FDs are trees[1] whose nodes denote features and whose edges represent top-down hierarchical decomposition of features. Each decomposition tells that, given the presence of the parent feature in some configuration c, some combination of its children should be present in c, too. Which combinations are allowed depends on the type of the decomposition, that is, the Boolean operator associated to the parent. In addition to their tree-shaped backbone, FDs can also contain cross-cutting constraints (usually *requires* or *excludes*) as well as side constraints in a textual language such as propositional logic [1].

Given an FD, the *configuration* or *product derivation process* is the process of gradually making the choices defined in the FD with the purpose of determining the product that is going to be built. In a realistic development, the configuration process is a small project itself, involving many people and taking up to several months [11]. In order to master the complexity of the configuration process, Czarnecki *et al.* [5] proposed the concept of *multi-level staged configuration* (MLSC), in which configuration is carried out by different stakeholders at different levels of product development or customisation. In simple staged configuration, at each stage some variability is removed from the FD until none is left. MLSC generalises this idea to the case were a set of related FDs are configured, each FD pertaining to a so-called 'level'. This addresses problems that

*FNRS Research Fellow.

[1] Sometimes DAGs are used, too [8].

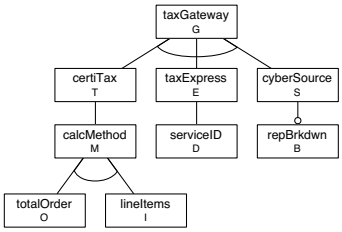


Figure 1. FD example, adapted from [5].

Table 1. FD decomposition operators

Concrete syntax	Boolen operator	Cardinality
	$and : \wedge$	$\langle n..n \rangle$
	$or : \vee$	$\langle 1..n \rangle$
	$xor : \oplus$	$\langle 1..1 \rangle$
		$\langle i..j \rangle$

occur when different abstraction levels are present in the same FD and also allows for more realism since a realistic project would have several related FDs rather than a single big one [12, 11].

Even though its supporting language (cardinality based FDs) received various formal semantics [4, 13], the MLSC process never received one. Nonetheless, a formal semantics is the primary indicator for precision and unambiguity and an important prerequisite for reliable tool-support. This paper is intended to fill this gap with a semantics for MLSC that builds on our earlier work on formal semantics for FDs [13]. The earlier semantics of [13] will be herein referred to as *static*, because it concentrates on telling which configurations are allowed (and which are disallowed), regardless of the process to be followed for reaching one or the other configuration. We thus extend this semantics with the concepts of *stage*, *configuration path* and *level*.

The contribution of the paper is a precise and formal account of MLSC that makes the original definition [5] more explicit and reveals some of its subtleties and incompletenesses. The semantics also allowed us to discover some important properties that an MLSC process should possess and a configuration tool should guarantee.

The paper is structured as follows. Section 2 recalls the static FD semantics and introduces a running example. Section 3 recapitulates the main concepts of staged configuration which are then formalised in Section 4 with the introduction of the dynamic semantics. Ways to implement and otherwise use the semantics are discussed in Section 5. The paper will be concluded in Section 6. An extended version of this paper was published as a technical report [3].

2 Static FD semantics ($[\![.]\!]_{FD}$)

In [13], we gave a general formal semantics to a wide range of FD dialects. The full details of the formalisation cannot be reproduced here, but we need to recall the essentials.[2] The formalisation was performed following the guidelines of Harel and Rumpe [7], according to whom each modelling language L must possess an unambiguous mathematical definition of three distinct elements: the *syntactic domain* $\mathcal{L}_L$, the *semantic domain* $\mathcal{S}_L$ and the *semantic function* $\mathcal{M}_L : \mathcal{L}_L \rightarrow \mathcal{S}_L$, also traditionally written $[\![\cdot]\!]_L$.

Our FD language will be simply called FD, and its syntactic domain is defined as follows.

Definition 1 (Syntactic domain $\mathcal{L}_{FD}$) *$d \in \mathcal{L}_{FD}$ is a 6-tuple $(N, P, r, \lambda, DE, \Phi)$ such that:*

- *N is the (non empty) set of features (nodes).*
- *$P \subseteq N$ is the set of primitive features.*
- *$r \in N$ is the root.*
- *$DE \subseteq N \times N$ is the decomposition relation between features which forms a tree. For convenience, we will use $children(f)$ to denote $\{g \mid (f, g) \in DE\}$, the set of all direct sub-features of f, and write $n \rightarrow n'$ sometimes instead of $(n, n') \in DE$.*
- *$\lambda : N \rightarrow \mathbb{N} \times \mathbb{N}$ indicates the decomposition type of a feature, represented as a cardinality $\langle i..j \rangle$ where i indicates the minimum number of children required in a product and j the maximum. For convenience, special cardinalities are indicated by the Boolean operator they represent, as shown in Table 1.*
- *Φ is a formula that captures crosscutting constraints ($\ll requires \gg$ and $\ll includes \gg$) as well as textual constraints. Without loss of generality, we consider Φ to be a conjunction of Boolean formulae on features, i.e. $\Phi \in \mathbb{B}(N)$, a language that we know is expressively complete wrt. $\mathcal{S}_{FD}$ [14].*

Furthermore, each $d \in \mathcal{L}_{FD}$ must satisfy the following well-formedness rules:

- *r is the root: $\forall n \in N(\nexists n' \in N \bullet n' \rightarrow n) \Leftrightarrow n = r$,*
- *DE is acyclic: $\nexists n_1, .., n_k \in N \bullet n_1 \rightarrow .. \rightarrow n_k \rightarrow n_1$,*
- *Terminal nodes are $\langle 0..0 \rangle$-decomposed.*

Definition 1 is actually a formal definition of the graphical syntax of an FD such as the one shown in Figure 1;

[2] Some harmless simplifications are made wrt. the original [13].

for convenience, each feature is given a name and a one-letter acronym. The latter depicts an FD for the tax gateway component of an e-Commerce system [5]. The component performs the calculation of taxes on orders made with the system. The customer who is going to buy such a system has the choice of three tax gateways, each offering a distinct functionality. Note that the hollow circle above feature B is syntactic sugar, expressing the fact that the feature is optional. In $\mathcal{L}_{FD}$, an optional feature f is encoded with a dummy (i.e. non-primitive) feature d that is $\langle 0..1 \rangle$-decomposed and having f as its only child [13]. Let us call B_d the dummy node inserted between B and its parent. The diagram itself can be represented as an element of $\mathcal{L}_{FD}$ where $N = \{G, T, E, ...\}$, $P = N \setminus \{B_d\}$, $r = G$, $E = \{(G, T), (G, E), ...\}$, $\lambda(G) = \langle 1..1 \rangle$, ... and $\Phi = \emptyset$.

The semantic domain formalises the real-world concepts that the language models, and that the semantic function associates to each diagram. FDs represent SPLs, hence the following two definitions.

Definition 2 (Semantic domain $\mathcal{S}_{FD}$) *$\mathcal{S}_{FD} \triangleq \mathcal{PPP}$, indicating that each syntactically correct diagram should be interpreted as a product line, i.e. a set of* configurations *or* products *(set of sets of primitive features).*

Definition 3 (Semantic function $[\![d]\!]_{FD}$) *Given $d \in \mathcal{L}_{FD}$, $[\![d]\!]_{FD}$ returns the* valid feature combinations *$FC \in \mathcal{PPN}$ restricted to primitive features: $[\![d]\!]_{FD} = FC_{|P}$, where the valid feature combinations FC of d are those $c \in \mathcal{PN}$ that:*

- *contain the root: $r \in c$,*
- *satisfy the decomposition type: $f \in c \wedge \lambda(f) = \langle m..n \rangle \Rightarrow m \leq |children(f) \cap c| \leq n$,*
- *justify each feature: $g \in c \wedge g \in children(f) \Rightarrow f \in c$,*
- *satisfy the additional constraints: $c \models \Phi$.*

The reduction operator used in Definition 3 will be used throughout the paper; it is defined as follows.

Definition 4 (Reduction $A_{|B}$)

$$A_{|B} \triangleq \{a' | a \in A \wedge a' = a \cap B\} = \{a \cap B | a \in A\}$$

Considering the previous example, the semantic function maps the diagram of Figure 1 to all its valid feature combinations, i.e. $\big\{\{G, T, M, O\}, \{G, T, M, I\}, ...\big\}$.

As shown in [13], this language suffices to retrospectively define the semantics of most common FD languages. The language for which staged configuration was initially defined [5], however, cannot entirely be captured by the above semantics [14]. The concepts of *feature attribute*, *feature reference* and *feature cardinality*[3] are missing. Attributes can easily be added to the semantics [4], an exercise we leave for future work. Feature cardinalities, as used for the *cloning* of features, however, would require a major revision of the semantics [4].

Benefits, limitations and applications of the above semantics have been discussed extensively elsewhere [13]. We just recall here that its main advantages are the fact that it gives an unambiguous meaning to each FD, and makes FDs amenable to automated treatment. The benefit of defining a semantics before building a tool is the ability to reason about tasks the tool should do on a pure mathematical level, without having to worry about their implementation. These so-called decision problems are mathematical properties defined on the semantics that can serve as indicators, validity or satisfiability checks.

In the present case, for instance, an important property of an FD, its *satisfiability* (i.e. whether it admits at least one product), can be mathematically defined as $[\![d]\!]_{FD} \neq \emptyset$. As we will see later on, the lack of formal semantics for staged configuration makes it difficult to precisely define such properties.

For the remainder of the paper, unless otherwise stated, we always assume d to denote an FD, and $(N, P, r, \lambda, DE, \Phi)$ to denote the respective elements of its abstract syntax.

3 Multi-level staged configuration

According to the semantics introduced in the previous section, an FD basically describes which configurations are allowed in the SPL, regardless of the *configuration process* to be followed for reaching one or the other configuration. Still, such a process is an integral part of SPL application engineering. According to Rabiser *et al.* [11], for instance, the configuration process generally involves many people and may take up to several months.

Czarnecki *et al.* acknowledge the need for explicit process support, arguing that in contexts such as *"software supply chains, optimisation and policy standards"*, the configuration is carried out in *stages* [5]. According to the same authors, a stage can be defined *"in terms of different dimensions: phases of the product lifecycle, roles played by participants or target subsystems"*. In an effort to make this explicit, they propose the concept of *multi-level staged configuration* (MLSC).

The principle of staged configuration is to remove part of the variability at each stage until only one configuration, the final product, remains. In [5], the refinement itself is achieved by applying a series of syntactic transformations

[3] Czarnecki *et al.* [5] distinguish *group* and *feature cardinalities*. Group cardinalities immediately translate to our decomposition types and $\langle 0..1 \rangle$ feature cardinalities to optional features. The $\langle i..k \rangle$ feature cardinalities, with $i \geq 0$ and $k > 1$, however, cannot be encoded in $\mathcal{L}_{FD}$.

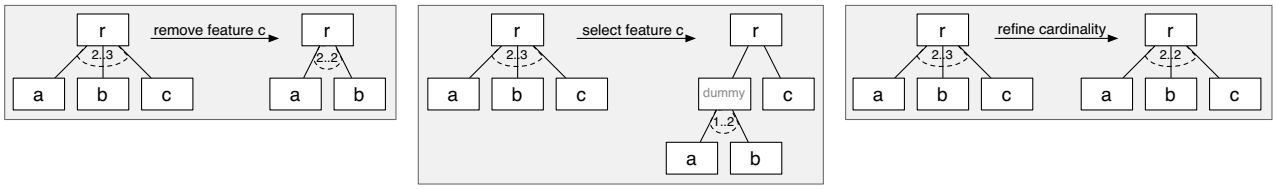


Figure 2. Specialisation steps, adapted from [5].

to the FD. Some of these transformations, such as setting the value of an attribute, involve constructs that are not formalised as part of the semantics defined in Section 2. The remaining transformations are show in Figure 2. Note that they are expressed so that they conform to our semantics.

Multi-level staged configuration is the application of this idea to a series of related FDs $d_1, .., d_\ell$. Each level has its own FD, and, depending on how they are linked, the configuration of one level will induce an automatic specialisation of the next level's FD. The links between diagrams are defined explicitly through *specialisation annotations*. A specialisation annotation of a feature f in d_i, $(f \in N_i)$, consists of a Boolean formulae ϕ over the features of $d_{i-1} (\phi \in \mathbb{B}(N_{i-1}))$. Once level $i-1$ is configured, ϕ can be evaluated on the obtained configuration $c \in [\![d_{i-1}]\!]_{FD}$, using the now standard Boolean encoding of [1], i.e. a feature variable n in ϕ is $true$ iff $n \in c$. Depending on its value and the specialisation type, the feature f will either be removed or selected through one of the first two syntactic transformations of Figure 2. An overview of this is shown in Table 2.

Let us illustrate this on the example of the previous section: imagine that there are two times at which the customer needs to decide about the gateways. The first time (level one) is when he purchases the system. All he decides at this point is which gateways will be available for use; the diagram that needs to be configured is the one shown on the left of Figure 3. Then, when the system is being deployed (level two), he will have to settle for one of the gateways and provide additional configuration parameters, captured by the first diagram on the right side of Figure 3. Given the inter-level links, the diagram in level two is automatically specialised based on the choices made in level one.

Note that even though both diagrams in the example are very similar, they need not be so. Also note that the original paper mentions the possibility, that several configuration levels might run in parallel. It applies, for instance, if levels represent independent decisions that need to be taken by different people. As we show later on, such situations give rise to interesting decision problems.

Finally, note that the MLSC approach, as it appears in [5], is entirely based on *syntactic* transformations. This makes it difficult to decide things such as whether two levels A and B are commutative (executing A before B leaves the same variability as executing B before A). This is the main motivation for defining a formal semantics, as follows in the next section.

4 Dynamic FD semantics ($[\![.]\!]_{CP}$)

We introduce the dynamic FD semantics in two steps. The first, Section 4.1, defines the basic staged configuration semantics; the second, Section 4.2, adds the multi-level aspect.

4.1 Staged configuration semantics

Since we first want to model the different stages of the configuration process, regardless of levels, the syntactic domain $\mathcal{L}_{FD}$ will remain as defined in Section 2. The semantic domain, however, changes since we want to capture the idea of building a product by deciding incrementally which configuration to retain and which to exclude.

Indeed, we consider the semantic domain to be the set of all possible *configuration paths* that can be taken when building a configuration. Along each such path, the initially full *configuration space* ($[\![d]\!]_{FD}$) progressively shrinks (i.e., configurations are discarded) until only one configuration is left, at which point the path stops. Note that in this work, we thus assume that we are dealing with *finite* configuration processes where, once a unique configuration is reached, it remains the same for the rest of the life of the application. Extensions of this semantics, that deal with reconfigurable systems, are discussed in [3]. For now, we stick to Definitions 5 and 7 that formalise the intuition we just gave.

Definition 5 (Dynamic semantic domain $\mathcal{S}_{CP}$) *Given a finite set of features N, a* configuration path *π is a finite sequence $\pi = \sigma_1 ... \sigma_n$ of length $n > 0$, where each $\sigma_i \in \mathcal{PP}N$ is called a* stage. *If we call the set of such paths C, then $\mathcal{S}_{CP} = \mathcal{P}C$.*

The following definition will be convenient when expressing properties of configuration paths.

Table 2. Possible inter-level links; original definition [5] left, translation to FD semantics right.

Specialisation type	Condition value	Specialisation operation	Equivalent Boolean constraint with $f \in N_i, \phi \in \mathbb{B}(N_{i-1}), c \in [\![d_{i-1}]\!]_{FD}$	
positive	*true*	select	$\phi(c) \Rightarrow f$	*Select* f, i.e. Φ_i becomes
positive	*false*	none		$\Phi_i \cup \{f\}$, if $\phi(c)$ is true.
negative	*false*	remove	$\neg\phi(c) \Rightarrow \neg f$	*Remove* f, i.e. Φ_i becomes
negative	*true*	none		$\Phi_i \cup \{\neg f\}$, if $\phi(c)$ is false.
complete	*true*	select	$\phi(c) \Leftrightarrow f$	*Select or remove* f depending on
complete	*false*	remove		the value of $\phi(c)$.

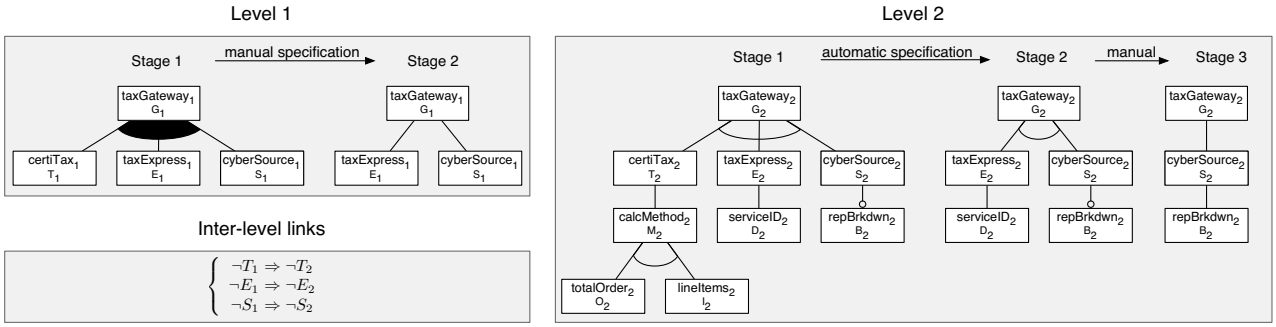


Figure 3. Example of MLSC, adapted from [5].

Definition 6 (Path notation and helpers)

- ϵ *denotes the empty sequence*
- $last(\sigma_1...\sigma_k) = \sigma_k$

Definition 7 (Staged configuration semantics $[\![d]\!]_{CP}$)
Given an FD $d \in \mathcal{L}_{FD}$, $[\![d]\!]_{CP}$ *returns all* legal *paths* π *(noted* $\pi \in [\![d]\!]_{CP}$*, or* $\pi \models_{CP} d$*) such that*

(7.1) $\sigma_1 = [\![d]\!]_{FD}$

(7.2) $\forall i \in \{2..n\} \bullet \sigma_i \subset \sigma_{i-1}$

(7.3) $|\sigma_n| = 1$

Note that this semantics is not meant to be used as an implementation directly, for it would be very inefficient. This is usual for denotational semantics which are essentially meant to serve as a conceptual foundation and a reference for checking the conformance of tools [15]. Along these lines, we draw the reader's attention to condition (7.2) which will force compliant configuration tools to let users make only "useful" configuration choices, that is, choices that effectively eliminate configurations. At the same time, tools must ensure that a legal product eventually remains reachable given the choices made, as requested by condition (7.3).

As an illustration, Figure 4 shows an example FD and its legal paths. A number of properties can be derived from the above definitions.

Theorem 8 (Properties of configuration paths)

(8.1) $[\![d]\!]_{FD} = \emptyset \Leftrightarrow [\![d]\!]_{CP} = \emptyset$

(8.2) $\forall c \in [\![d]\!]_{FD} \bullet \exists \pi \in [\![d]\!]_{CP} \bullet last(\pi) = \{c\}$

(8.3) $\forall \pi \in [\![d]\!]_{CP} \bullet \exists c \in [\![d]\!]_{FD} \bullet last(\pi) = \{c\}$

Contrary to what intuition might suggest, (8.2) and (8.3) do not imply that $|[\![d]\!]_{FD}| = |[\![d]\!]_{CP}|$, they merely say that every configuration allowed by the FD can be reached as part of a configuration path, and that each configuration path ends with a configuration allowed by the FD.

Czarnecki *et al.* [5] define a number of transformation rules that are to be used when specialising an FD, three of which are shown in Figure 2. With the formal semantics, we can now verify whether these rules are expressively complete, i.e. whether is it always possible to express a σ_i $(i > 1)$ through the application of the three transformation rules.

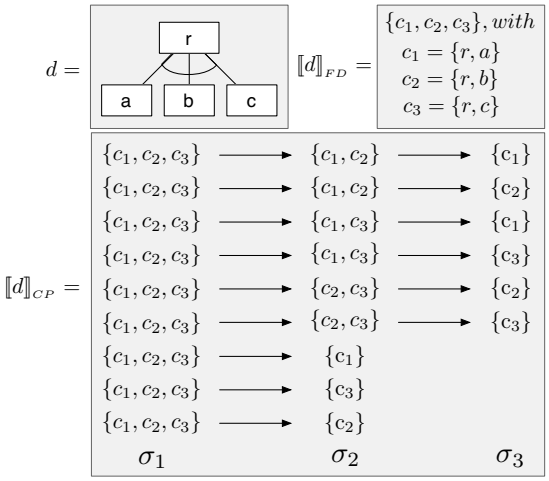


Figure 4. The staged configuration semantics illustrated.

Theorem 9 (Incompleteness of transformation rules) *The transformation rules shown in Figure 2 are expressively incomplete wrt. the semantics of Definition 7.*

Proof. Consider a diagram consisting of a parent feature $\langle 2..2 \rangle$-decomposed with three children a, b, c. It is not possible to express the σ_i consisting of $\{a, b\}$ and $\{b, c\}$, by starting at $\sigma_1 = \{\{a, b\}, \{a, c\}, \{b, c\}\}$ and using the proposed transformation rules (since removing one feature will always result in removing at least two configurations). □

Note that this is not necessarily a bad thing, since Czarnecki *et al.* probably chose to only include transformation steps that implement the most frequent usages. However, the practical consequences of this limitation need to be assessed empirically.

4.2 Adding levels

Section 4.1 only deals with dynamic aspects of staged configuration of a single diagram. If we want to generalise this to MLSC, we need to consider multiple diagrams and links between them. To do so, there are two possibilities: (1) define a new abstract syntax, that makes the set of diagrams and the links between them explicit, or (2) encode this information using the syntax we already have.

We chose the latter option, mainly because it allows to reuse most of the existing definitions and infrastructure, and because it can more easily be generalised. Indeed, a set of FDs, linked with conditions of the types defined in Table 2, can be represented as a single big FD. The root of each individual FD becomes a child of the root of the combined FD.

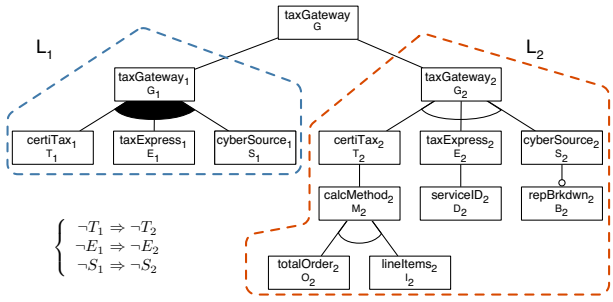


Figure 5. Example of Figure 3 in $\mathcal{L}_{DynFD}$.

The root is *and*-decomposed and the inter-level links are represented by Boolean formulae. To keep track of where the features in the combined FD came from, the level information will be made explicit as follows.

Definition 10 (Dynamic syntactic domain $\mathcal{L}_{DynFD}$) *$\mathcal{L}_{DynFD}$ consists of 7-tuples $(N, P, L, r, \lambda, DE, \Phi)$, where:*

- *$N, P, r, \lambda, DE, \Phi$ follow Definition 1,*
- *$L = L_1 ... L_\ell$ is a partition of $N \setminus \{r\}$ representing the list of levels.*

So that each $d \in \mathcal{L}_{DynFD}$ satisfies the well-formedness rules of Definition 1, has an and-decomposed root, and each level $L_i \in L$:

- *is connected through exactly one node to the global root: $\exists! n \in L_i \bullet (r, n) \in DE$, noted hereafter $root(L_i)$,*
- *does not share decomposition edges with other levels (except for the root): $\forall (n, n') \in DE \bullet (n \in L_i \Leftrightarrow n' \in L_i) \vee (n = r \wedge n' = root(L_i))$,*
- *is itself a valid FD, i.e. $(L_i,\ P \cap L_i,\ root(L_i),\ \lambda \cap (L_i \rightarrow \mathbb{N} \times \mathbb{N}),\ DE \cap (L_i \times L_i),\ \emptyset)$ satisfies Definition 1.*[4]

Figure 5 illustrates how the example of Figure 3 is represented in $\mathcal{L}_{DynFD}$. Note that, for the purpose of this paper, we chose an arbitrary concrete syntax for expressing levels, viz. the dotted lines. This is meant to be illustrative, since a tool implementation should rather present each level separately, so as to not harm scalability.

Given the new syntactic domain, we need to revise the semantic function. As for the semantic domain, it can remain the same, since we still want to reason about the possible configuration paths of an FD. The addition of multiple

[4] The set of constraints here is empty because it is not needed for validity verification.

levels, however, requires us to reconsider what a *legal* configuration path is. Indeed, we want to restrict the configuration paths to those that obey the levels specified in the FD. Formally, this is defined as follows.

Definition 11 (Dynamic FD semantics $[\![d]\!]_{DynFD}$) *Given an FD $d \in \mathcal{L}_{DynFD}$, $[\![d]\!]_{DynFD}$ returns all paths π that are legal wrt. Definition 7, i.e. $\pi \in [\![d]\!]_{CP}$, and for which exists a* legal level arrangement*, that is π, except for its initial stage, can be divided into ℓ $(= |L|)$* levels*: $\pi = \sigma_1\Sigma_1..\Sigma_\ell$, each Σ_i corresponding to an L_i such that:*

(11.1) Σ_i *is fully configured:* $|final(\Sigma_i)_{|L_i}| = 1$*, and*

(11.2) $\forall \sigma_j\sigma_{j+1} \bullet \pi = ...\sigma_j\sigma_{j+1}...$ *and* $\sigma_{j+1} \in \Sigma_i$*, we have*

$$(\sigma_j \setminus \sigma_{j+1})_{|L_i} \subseteq (\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i}).$$

As before, this will be noted $\pi \in [\![d]\!]_{DynFD}$, or $\pi \models_{DynFD} d$.

We made use of the following helper.

Definition 12 (Final stage of a level Σ_i) *For $i = 1..\ell$,*

$$final(\Sigma_i) \triangleq \begin{cases} last(\Sigma_i) & \text{if } \Sigma_i \neq \epsilon \\ final(\Sigma_{i-1}) & \text{if } \Sigma_i = \epsilon \text{ and } i > 1 \\ \sigma_1 & \text{if } \Sigma_i = \epsilon \text{ and } i = 1 \end{cases}$$

The rule (11.2) expresses the fact that each configuration deleted from σ_j (i.e. $c \in \sigma_j \setminus \sigma_{j+1}$) during level L_i must be necessary to delete one of the configurations of L_i that are deleted during this stage. In other words, the set of *deleted* configurations needs to be included in the set of *deletable* configurations for that level. The deletable configurations in a stage of a level are those that indeed remove configurations pertaining to that level (hence: first reduce to the level, then subtract), whereas the deleted configurations in a stage of a level are all those that were removed (hence: first subtract, then reduce to level to make comparable). Intuitively, this corresponds to the fact that each decision has to affect only the level at which it is taken.

4.3 Illustration

Let us illustrate this with the FD of Figure 5, which we will call d, itself being based on the example of Figure 3 in Section 3. The semantic domain of $[\![d]\!]_{DynFD}$ still consists of configuration paths, i.e. it did not change from those of $[\![d]\!]_{CP}$ shown in Figure 4. Yet, given that $[\![d]\!]_{DynFD}$ takes into account the levels defined for d, not all possible configuration paths given by $[\![d]\!]_{CP}$ are legal. Namely, those that do not conform to rules (11.1) and (11.2) need to be discarded. This is depicted in Figure 6, where the upper box denotes the staged configuration semantics of d ($[\![d]\!]_{CP}$), and the lower box denotes $[\![d]\!]_{DynFD}$, i.e. the subset of $[\![d]\!]_{CP}$ that conforms to Definition 11.

Table 3. Validation of level arrangements.

Level arrangement for path	rule (11.1)	rule (11.2)
$\pi_i = \sigma_1$ Σ_1 Σ_2 / Σ_1 Σ_2 / Σ_1 Σ_2 / σ_{2_i} σ_{3_i}	FALSE	/
	TRUE	FALSE
	TRUE	TRUE
$\pi_j = \sigma_1$ Σ_1 Σ_2 / Σ_1 Σ_2 / Σ_1 Σ_2 / σ_{2_j} σ_{3_j}	FALSE	/
	TRUE	FALSE
	TRUE	FALSE

We now zoom in on two configuration paths $\pi_i, \pi_j \in [\![d]\!]_{CP}$, shown with the help of intermediate FDs in the lower part of Figure 6. As noted in Figure 6, π_j is not part of $[\![d]\!]_{DynFD}$ since it violates Definition 11, whereas π_i satisfies it and is kept. The rationale for this is provided in Table 3. Indeed, for π_j, there exists no level arrangement that would satisfy both rules (11.1) and (11.2). This is because in σ_{2_j}, it is not allowed to remove the feature B_2, since it belongs to L_2, and L_1 is not yet completed. Therefore, either there is still some variability left in the FD at the end of the level, which is thus not fully configured (the first possible arrangement of π_j in Table 3 violates rule (11.1)), or the set of deleted configurations is greater than the set of deletable configurations (the other two arrangements of π_j in Table 3, which violate rule (11.2)). For π_i, on the other hand, a valid level arrangement exists and is indicated by the highlighted line in Table 3. More details for this illustration are provided in [3].

5 Towards automation and analysis

This section explores properties of the semantics we just defined and sketches paths towards automation.

5.1 Properties of the semantics

In Definition 11, we require that it has to be possible to divide a configuration path into level arrangements that satisfy certain properties. The definition being purely declarative, it does not allow an immediate conclusion as to how many valid level arrangements one might find. The following two theorems show that there is exactly one. Their proofs can be found in [3].

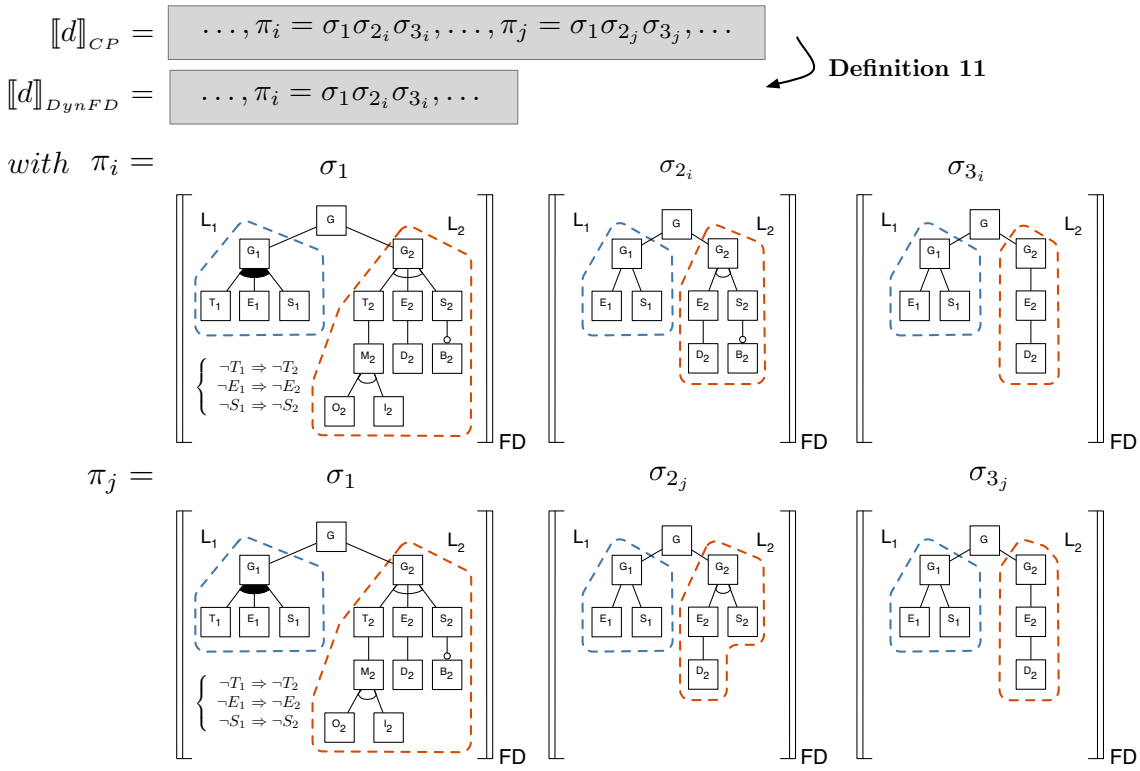


Figure 6. Example of Figure 3 in $[\![d]\!]_{CP}$ and $[\![d]\!]_{DynFD}$.

Theorem 13 (Properties of level arrangements) *Given a diagram $d \in \mathcal{L}_{DynFD}$, each configuration path $\pi \in [\![d]\!]_{DynFD}$ with $\Sigma_1..\Sigma_\ell$ as a valid level arrangement satisfies the following properties.*

(13.1) If $\sigma_j \in \Sigma_i$ then $\forall k < j \bullet |\sigma_{k\,|L_i}| > |\sigma_{j\,|L_i}|$.

(13.2) If $\sigma_j \in \Sigma_i$ and $\sigma_j \neq last(\Sigma_i)$ then $|\sigma_{j\,|L_i}| > 1$.

(13.3) If $|\sigma_{j\,|L_i}| = 1$ then $\forall k > j \bullet \sigma_k \notin \Sigma_i$.

(13.4) If $|\sigma_{j\,|L_i}| = 1$ then $\forall k > j \bullet |\sigma_{k\,|L_i}| = 1$.

Theorem 14 (Uniqueness of level arrangement) *For any diagram $d \in \mathcal{L}_{DynFD}$, a level arrangement for a configuration path $\pi \in [\![d]\!]_{DynFD}$ is unique.*

An immediate consequence of this result is that it is possible to determine a legal arrangement *a posteriori*, i.e. given a configuration path, it is possible to determine a unique level arrangement describing the process followed for its creation. Therefore, levels need not be part of the semantic domain. This result leads to the following definition.

Definition 15 (Subsequence of level arrangement) *Given an FD d and $L_i \in L, \pi \in [\![d]\!]_{DynFD}$, $sub(L_i, \pi)$ denotes the subsequence Σ_i of π pertaining to level L_i for the level arrangement of π that satisfies Definition 11.*

Continuing with Definition 11, remember that rule (11.2) requires that every *deleted* configuration be *deletable* in the stage of the associated level. An immediate consequence of this is that, unless we have reached the end of the configuration path, the set of *deletable* configurations must not be empty, established in Theorem 16. A second theorem, Theorem 17, shows that configurations that are deletable in a stage, are necessarily deleted in this stage.

Theorem 16 *A necessary, but not sufficient replacement for rule (11.2) is that $(\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i}) \neq \emptyset$.*

Proof. Immediate via *reductio ad absurdum.* □

Theorem 17 *For rule (11.2) of Definition 11 holds*

$$(\sigma_j \setminus \sigma_{j+1})_{|L_i} \subseteq (\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i}) \Rightarrow (\sigma_j \setminus \sigma_{j+1})_{|L_i} = (\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i}).$$

Proof. In [3], we prove that always

$$(\sigma_j \setminus \sigma_{j+1})_{|L_i} \supseteq (\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i}).$$

which means that if in addition $(\sigma_j \setminus \sigma_{j+1})_{|L_i} \subseteq (\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i})$ holds, both sets are equal. □

In Theorem 9, Section 4.1, we showed that the transformation rules of Figure 2, i.e. those proposed in [5] that relate to constructs formalised in the abstract syntax of Definition 10, are not expressively complete wrt. the basic staged configuration semantics of Definition 7. The two following theorems provide analogous results, but for the dynamic FD semantics. Basically, the property still holds for the dynamic FD semantics of Definition 11, and a similar property holds for the proposed inter-level link types of Table 2.

Theorem 18 (Incompleteness of transformation rules) *The transformation rules shown in Figure 2 are expressively incomplete wrt. the semantics of Definition 11.*

Proof. We can easily construct an example for $\mathcal{L}_{DynFD}$; it suffices to take the FD used to prove Theorem 9 and to consider it as the sole level of a diagram. From there on, the proof is the same. □

Theorem 19 (Incompleteness of inter-level link types) *The inter-level link types proposed in [5] are expressively incomplete wrt. the semantics of Definition 11.*

Proof. Basically, the proposed inter-level link types always have a sole feature on their right-hand side. It is thus impossible, for example, to express the fact that if some condition ϕ is satisfied for level L_i, all configurations of level L_{i+1} that have f will be excluded if they also have f' (i.e. $\phi \Rightarrow (f' \Rightarrow \neg f)$). □

5.2 Implementation strategies

A formal semantics is generally the first step towards an implementation, serving basically as a specification. In the case of FDs, two main types of tools can be considered: *modelling* tools, used for creating FDs, and *configuration* tools, used during the product derivation phase. Since the only difference between $\mathcal{L}_{FD}$ and $\mathcal{L}_{DynFD}$ is the addition of configuration levels, it should be rather straightforward to extend existing FD modelling tools to $\mathcal{L}_{DynFD}$. In addition, the core of the presented semantics deals with configuration. Let us therefore focus on how to implement a configuration tool for $\mathcal{L}_{DynFD}$, i.e. a tool that allows a user to configure a feature diagram $d \in \mathcal{L}_{DynFD}$, allowing only the configuration paths in $[\![d]\!]_{DynFD}$, and preferably without having to calculate the whole of $[\![d]\!]_{FD}$, $[\![d]\!]_{CP}$ or $[\![d]\!]_{DynFD}$. Also note that, since we do not consider ourselves experts in human-machine interaction, we restrict the following discussion to the implementation of the semantics independently from the user interface. It goes without saying that at least the same amount of thought needs to be devoted to this activity [2].

The foundation of a tool, except for purely graphical ones, is generally a reasoning back-end. Mannion and Batory [9, 1] have shown how an FD d can be encoded as a Boolean formula, say $\Gamma_d \in \mathbb{B}(N)$; and a reasoning tool based on this idea exists for $\mathcal{L}_{FD}$ [16]. The free variables of Γ_d are the features of d, so that, given a configuration $c \in [\![d]\!]_{FD}$, $f_i = true$ denotes $f_i \in c$ and $false$ means $f_i \notin c$. The encoding of d into Γ_d is such that evaluating the truth of an interpretation c in Γ_d is equivalent to checking whether $c \in [\![d]\!]_{FD}$. More generally, satisfiability of Γ_d is equivalent to non-emptiness of $[\![d]\!]_{FD}$. Given this encoding, the reasoning back-end will most likely be a SAT solver, or a derivative thereof, such as a logic truth maintenance system (LTMS) [6] as suggested by Batory [1].

The configuration tool mainly needs to keep track of which features were selected, which were deselected and what other decisions, such as restricting the cardinality of a decomposition, were taken. This *configuration state* basically consists in a Boolean formula $\Delta_d \in \mathbb{B}(N)$, that captures which configurations have been discarded. Feasibility of the current configuration state, i.e. whether all decisions taken were consistent, is equivalent to satisfiability of $\Gamma_d \wedge \Delta_d$. The configuration process thus consists in adding new constraints to Δ_d and checking whether $\Gamma_d \wedge \Delta_d$ is still satisfiable.

A tool implementing the procedure sketched in the previous paragraph will inevitably respect $[\![d]\!]_{FD}$. In order to respect $[\![d]\!]_{CP}$, however, the configuration tool also needs to make sure that each time a decision δ is taken, all other decisions implied by δ be taken as well, for otherwise rule (7.2) might be violated in subsequent stages. This can easily be achieved using an LTMS which can propagate constraints as the user makes decisions. This way, once she has selected a feature f that excludes a feature f', the choice of f' will not be presented to the user anymore. The LTMS will make it easy to determine which variables, i.e. features, are still free and the tool should only present those to the user.

The extended procedure would still violate $[\![d]\!]_{DynFD}$, since it does not enforce constraints that stem from level definitions. A second extension is thus to make sure that the tool respects the order of the levels as defined in d, and only presents choices pertaining to the current level L_i until it is dealt with. This means that the formula of a decision δ may only involve features f that are part of the current level (rule (11.2)). It also means that the tool needs to be able to detect when the end of a level L_i has come (rule (11.1)), which is equivalent to checking whether, in the current state of the LTMS, all of the $f \in L_i$ are assigned a fixed value.

Given these guidelines, it should be relatively straightforward to come up with an architecture and some of the principal algorithms for a tool implementation.

6 Conclusion and future work

We introduced a dynamic formal semantics for FDs that allows reasoning about its configuration paths, i.e. the con-

figuration process, rather than only about its allowed configurations. Extending the basic dynamic semantics with levels yields a semantics for MLSC. The contribution of the paper is therefore a precise and formal account of MLSC that makes the original definition [5] more explicit and reveals some of its subtleties and incompletenesses. Based on the semantics we show some interesting properties of configuration paths and outline an implementation strategy that uses SAT solvers as the reasoning back-end.

A number of extensions to the dynamic FD semantics can be envisioned. From the original definition of MLSC [5], it inherits the assumption that levels are configured one after the other in a strict order until the final configuration is obtained. One way to extend the semantics is to relax this restriction and to allow levels that are interleaved, or run in parallel. The semantics also assumes that the configuration finishes at some point. This is not the case for dynamic or self-adaptive systems. Those systems have variability left at runtime, allowing them to adapt to a changing environment. In this case, configuration paths would have to be infinite. Another extension we envision is to add new FD constructs (like feature cardinalities and attributes) to the formalism. The ultimate goal of our endeavour is naturally to develop a configurator that would be compliant with the formalism, verify properties and compute various indicators.

These points are partly discussed in Section 2 and more extensively in [3]. They will be elaborated on in our future work, where we also intend to tackle the problem of FD evolution taking place during configuration.

Acknowledgements

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

References

[1] D. S. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC'05*, pages 7–20, 2005.

[2] C. Cawley, D. Nestor, A. Preußner, G. Botterweck, and S. Thiel. Interactive visualisation to support product configuration in software product lines. In *VaMoS'08*, 2008.

[3] A. Classen, A. Hubaux, and P. Heymans. A formal semantics for multi-level staged configuration. Technical Report P-CS-TR SPLBT-00000002, PReCISE Research Center, University of Namur, Namur, Belgium, November 2008. Download at www.fundp.ac.be/pdf/publications/66426.pdf.

[4] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[5] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[6] K. Forbus and J. de Kleer. *Building Problem Solvers*. The MIT Press, 1993.

[7] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff - part I: The basic stuff. Technical Report MCS00-16, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Israel, September 2000.

[8] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annales of Software Engineering*, 5:143–168, 1998.

[9] M. Mannion. Using First-Order Logic for Product Line Model Validation. In *SPLC'02*, LNCS 2379, pages 176–187, San Diego, CA, Aug. 2002. Springer.

[10] K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, July 2005.

[11] R. Rabiser, P. Grunbacher, and D. Dhungana. Supporting product derivation by adapting and augmenting variability models. In *SPLC'07*, pages 141–150, Washington, DC, USA, 2007. IEEE Computer Society.

[12] M.-O. Reiser and M. Weber. Managing highly complex product families with multi-level feature trees. In *RE'06*, pages 146–155, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[13] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *RE'06*, pages 139–148, Minneapolis, Minnesota, USA, September 2006.

[14] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, page 38, 2006.

[15] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.

[16] J.-C. Trigaux and P. Heymans. Varied feature diagram (vfd) language: A reasoning tool. Technical Report EPH3310300R0462 / 215315, PRECISE, University of Namur, January 2007. PLENTY: Product Line ENgineering of food TraceabilitY software.

A Model for Trading off Flexibility and Variability in Software Intensive Product Development

Wim Codenie[1], Nicolás González-Deleito[1], Jeroen Deleu[1], Vladimir Blagojević[1], Pasi Kuvaja[2], and Jouni Similä[2]
[1]*Software Engineering Group, Sirris, Belgium*
[2]*Department of Information Processing Science, University of Oulu, Finland*
wim.codenie@sirris.be

Abstract

Flexible development and product variability are two different challenges experienced by an increasing number of software intensive product builders. These challenges are often mixed, and have different meanings depending on the development model used by software intensive product builders to develop a product. Furthermore, the context in which companies operate can force them to transition between software product development models, and to re-interpret flexibility and variability in the context of the new development model. This paper explains the difference between flexible development and product variability, and provides a tradeoff model to assist companies in improving flexibility and variability in the framework of their specific context.

1. Introduction

Flexible development (flexibility) and product variability (variability) have become strategic challenges in software intensive product development. Although they are often mixed, both are fundamentally different problems. *Flexible development* is the ability to efficiently respond to new market opportunities in an effective manner (i.e. the speed by which ideas are brought to the market). *Product variability* is the ability to offer a large number of product variants to customers in an efficient manner. For many software intensive product builders, being able to address one or even both of these challenges can result in a competitive advantage.

Given the large available state of the art in flexibility and variability management techniques, it is remarkable that implementing an effective flexibility and variability strategy still poses so many challenges to software intensive product builders [27] [12].

An empirical study conducted by Sirris at 57 software product builders in Belgium [12] reveals two root causes for these difficulties. A first reason is that companies often try to apply the state of the art in flexibility and variability "out of the box". In other words, they neglect to interpret the flexibility and variability challenges in the scope of their specific context, i.e. the product development model they use. Four basic product development models are considered in this paper: *project based development*, *technology platform development*, *customized product development* and *out of the box product development*.

A second reason is caused by changes in the business environment and in the technological context, driving companies to change their product strategy. Few products remain built using the same product development model for a long period of time. Instead, companies are undergoing transitions towards other development models. As a consequence, they need to re-interpret flexibility and variability each time the development model changes.

The goal of this paper is to provide insights in the dependence of flexibility and variability on software product development models. This paper presents a tradeoff model to assist companies in improving flexibility and variability, by selecting the optimal development model for flexibility and variability and/or optimizing flexibility and variability within the chosen model.

This paper is organized as follows. Section 2 introduces software intensive product development. Section 3 provides an overview of the four basic software product development models considered in this paper. Section 4 introduces flexibility and variability and provides an argumentation why they are becoming so dominant. It also presents an overview of

the state of the art in both domains. Section 5 introduces the static view on flexibility and variability: the different meaning of flexibility and variability depending on the product development model in use. Section 6 extends this to the dynamic view by observing that many companies are facing transitions between the software product development models requiring a re-interpretation of flexibility and variability. Section 7 introduces the core of the tradeoff model.

2. Software intensive product development

Product development is the set of activities starting with the perception of a market opportunity and ending in the production, sales, and delivery of a product [36]. Software is playing an increasingly important role in product development and has become a major instrument for product innovation [20] [30]. Because of this, software is ascending the value chain of products. This is true for many sectors, including sectors that traditionally are not associated with software. European studies [20] forecast an increase in the number of products that will become software enabled, and a strong global increase in the size of software development.

Throughout the paper, the term *software intensive product* is used to denote a product that contains a significant software component. Companies that produce these products are called *software intensive product builders* and the term *software intensive product development* is used to refer to product development as applied by software intensive product builders.

3. Software product development models

Companies use different models to develop and deliver their software intensive products to the market. At the current stage of the research, the four models described below are considered. They can be classified according to the ratio of *domain engineering* and *application engineering* they require (Figure 1, note that the relative position of the four models is only indicative). Domain engineering encompasses all the engineering activities related to the development of software artifacts reusable across a set of products (i.e. activities done for groups of customers such as making a general purpose product). Application engineering encompasses all the engineering activities related to the development of one, specific software product (i.e. all the effort done for a single customer such as developing a tailor made solution) [7] [10] [28].

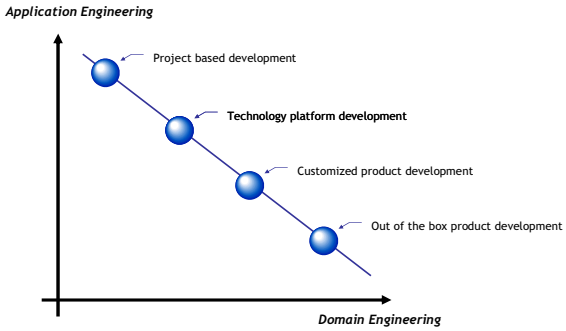


Figure 1. The four software product development models

Two other terms used in the description of the different models are *customization* and *configuration*. In this paper the term customization is used to refer to the activity of changing a (generic) product into a solution satisfying the specific needs of a customer (by either adding new functionality, or changing or removing existing functionality). In essence, customization creates a new product variant that did not exist before. Configuration is choosing among a predefined set of product variants (e.g. by filling in configuration parameters).

Project based development. This model is used when products are developed on a project basis. In this model, customers do not find the desired solution on the market and commission its development to a software intensive product builder. Products developed in this model aim therefore at addressing a very specific demand that needs to perfectly fit in the context of the customer. Software intensive product builders applying this model usually build these products from scratch, independently from their other possibly similar products. In some cases, excerpts from other projects might be used to build a new product. This results in ad-hoc reuse between projects. In this model, the main focus is on delivering on time, within budget and with a given quality level.

Technology platform development. In this model, software intensive product builders aim to build a dedicated platform that solves a specific technological problem (e.g. database persistence, transactional event handling, etc.). Platforms aim to be building blocks of larger products and can either be for internal use (i.e. to be included in other products developed by the platform builder) or for external use (i.e. to be included in products developed by other companies). Companies developing a platform for internal use do it because they need the technology, but find no solution on the market. On the other hand, companies developing a platform to be used in third party products are constantly searching for new opportunities for their

technology. Once such an opportunity is found, the challenge is to integrate the platform with other products as fast and efficiently as possible. Embedding the platform in the encapsulating product typically involves a high degree of integration activities from both the platform builder and the customer.

Customized product development. This model is used by software intensive product builders that develop products that need to be further adapted to suit the specific needs of customers. These companies typically operate in markets in which customers seem to have conflicting expectations. On the one hand, customers expect solutions with qualities associated with product development (e.g. proven robustness, immediate availability, proven stability, a sophisticated maintenance/support network, etc.). On the other hand, they expect solutions with qualities associated with tailor made project development (e.g. possibility of total customer solution, every request of the customer is 100% satisfied by the solution, etc.). In order to fulfill these expectations, product builders applying this model focus on developing a common product base (e.g. some kind of "semi-finished" product) that contains the core functionality shared across all the potential customers. This product base is used to derive products for customers (customization) – by either the product builder itself or by a third party. Characteristic of the markets these product builders are active in is that a high degree of commonalities can be found between customer needs (e.g. 80%) but at the same time, customers can have very different and opposing needs for the (smaller) remaining part (e.g. 20%).

Out of the box product development. In this model, software intensive product builders strive to make a generic product that can suit the needs of many customers. The difference with the customized product development model is that no customization (coding) is done by the product builder for individual customers. The assumption is that customers will be prepared to sacrifice features in favor of, for example, lower cost and product stability. For out of the box product builders, finding the optimal degree of product configurability is often a challenge. Configurability can take many forms: support for different regulations (e.g. region specific features, legislation), support for different platforms (e.g. Windows versus Linux), support for different license models (e.g. free edition, professional edition, home edition), or personalization of the product.

Software intensive product builders do not necessarily restrict themselves to only one of the above product development models. Companies that offer a rich product portfolio can apply a different software product development model for each product in the portfolio, depending on parameters such as market situation and product maturity.

Other classifications are described in the literature. Bosch [7] considers seven maturity levels for architecture-centric, intra-organization reuse in the context of software product lines. Cusumano [13] considers two kinds of companies: *platform leaders* and *platform complementors*.

A similarity between the above models and some of the maturity levels for software product lines defined by Bosch [7] can be observed. Unlike Bosch, the described four software product development models give no indication about maturity. For example, the out of the box product development model is not necessarily more mature than the project based development model, and vice versa. In a very small niche market (or even for one customer with very specific needs) a company may benefit to develop a product from scratch in a project based manner. This will be a valid strategy if customers are prepared to pay the premium price for getting exactly what they need and if little commonality exists between different customers' needs. Maturity must be interpreted differently for each model, e.g. some companies might be very mature in applying the project based development model while others might not.

4. Flexibility and variability in software intensive product development

For many software intensive product builders resolving flexibility and/or variability is becoming a necessity to remain competitive.

4.1. Product variability: the consequence of embracing customer intimacy

Driven by customers that are increasingly cost-conscious and demanding, a large number of companies adhere to *customer intimacy*: a strategy of continually tailoring and shaping products and services to fit an increasingly fine definition of the customer [35]. This has become a business strategy that is adopted by 1 out of 2 companies [23]. When applied on a large scale, customer intimacy leads to a trend called mass customization: producing in large volumes, but at the same time giving each individual customer something different. Achieving customer intimacy and especially mass customization leads in most cases to a large number of product variants per product base. This

phenomenon is called *product variability*, the ability to offer a large number of product variants to customers.

As an increasing number of products will become software and ICT enabled, the variability aspects of products will be more and more situated at the software level. This drives an increasing number of companies to raise the variability level in their software. Forecasts predict that the ability to implement an efficient and effective variability strategy will be an important prerequisite to succeed or even survive as a software intensive product builder [19] [23]. Unfortunately, many product builders do not reach the desired level of variability or fail to do so in a cost efficient manner. This confronts software intensive product builders with a challenging engineering paradox:

Variability Paradox: *How to remain efficient and effective while at the same time offer a much richer product variety to the customers?*

4.2. Flexible development: the consequence of embracing innovation

More and more the economy is evolving from a knowledge based economy to an economy based on creativity and innovation [24]. A study by Siemens illustrates that today up to 70% of the revenue of product builders is generated by products or product features that did not exist 5 years ago [30]. On top of this, 90% of the managers of companies in sectors like aviation, automotive, pharmaceutical industry and telecommunication consider innovation as essential to reach their strategic objectives [15]. The "Innovator's dilemma" [9] is in many cases no longer a dilemma for companies that build products, as innovation has become an absolute necessity in order to deal with global challenges and trends of the future.

An important observation to make in that context is that software is increasingly being used as an instrument to realize that innovation: software as engine for innovation. This trend is not only valid within specific niche markets, it is relevant for a wide range of sectors [20] [25]. Software is no longer a supporting technology, but it takes an essential role in the process of value creation.

Being "the first" is important if one wants to be innovative. Because of that, many companies are competing in a "rush to market" race. The product life cycle is shrinking at a steady pace, and few are the products today with a life cycle of one year or longer. The principle of being first mover, the rush to market race and the innovation dilemma lead to a need for flexibility.

Flexible development: *the ability to quickly respond to new market needs and customer requests. It is all about increasing the speed by which innovations and ideas are brought to the market. Because software is "invading" in more products, flexibility is becoming a major issue in software development.*

4.3. Flexibility and variability are different

Although flexibility and variability are very different by nature, they are often mixed. On the one hand, product variability is an attribute of the product (or a product base or a platform). Increasing the degree of variability of a product corresponds to adding variation points to that product. A *variation point* is a product option. It describes a decision that can be postponed. A variation point can be fixed by making a choice between a number of options. On the other hand, flexible development is an attribute of the engineering process. Being flexible basically means dealing with uncertainty, with the unexpected, and being able to efficiently adapt to changes and to new contexts.

Sometimes flexibility is increased by introducing variability. In [33] for example, modularity in product architecture is put forward as an anticipation strategy to become more flexible. The reverse (increasing variability by introducing flexibility) also happens. In Extreme programming [5] for instance, extensive upfront reasoning about future changes (with the purpose of anticipating variability) is discouraged in favor of installing a process that allows smoother software evolutions. The "extreme" view is that if you are flexible enough you do not need to be variable.

For some companies only variability matters, for others only flexibility matters, and for some others both matter. Also, companies exist that make products for which neither variability nor flexibility matter. A study conducted by Sirris at 57 companies [12] reveals that for 60% of the surveyed companies flexibility is important (i.e. the company experienced difficulties managing the challenge and it foresaw considerable optimizations if the challenge could be managed better), for 45% of companies variability is important and for 30% both challenges are important.

4.4. State of the art in product variability

Variability can be addressed from different perspectives. From the technology perspective, object-oriented frameworks [1], design patterns [16] and aspect-oriented development [22] have progressively been developed to provide better support for reuse and

variability. More recently, generative techniques [14] (a.k.a. model driven development [32]) aim to generate code in an automated way from high-level specifications. Some of these techniques can be used also to generate product variants from a common product.

Another relevant research area is the domain of configuration techniques. This ranges from simple configuration strategies based on parameterization (e.g. XML) up to fundamental research in configuration languages, consisting in using domain specific languages for configuration [26]. The goal is to extend products with languages to allow more complex configurations than is possible with conventional configuration parameters.

From the engineering perspective, various modeling techniques for variability have been proposed both by academia and industry [28]. Many of them have their roots in feature modeling (e.g. FODA), but significant contributions also have come from related areas including configuration management, domain analysis, requirements engineering, software architecture, formal methods, and domain specific languages [21] [4].

Finally, software product lines [10] have received extensive attention during the last few years (an issue of CACM has even been devoted to that topic [2]), and an increasing number of companies are considering adopting them.

4.5. State of the art in flexible development

Flexible development, the ability to quickly respond to new market needs and customer requests, is typically achieved through agile development methods. These methods basically promote communication between individuals, collaboration with customers, and frequent product increments with business value [6] [3]. Examples of popular agile methods are Scrum [31] and Extreme programming [5]. Scrum focuses mostly on agile management mechanisms. This is in contrast to Extreme programming for example, where the focus is more on providing integrated software engineering practices.

Another approach to achieve flexibility is Lean development [29]. It aims at improving the development process by removing any activity not providing value (waste elimination).

Finally, another view presented in the literature is that flexibility can be achieved by creating (flexible) adaptable artifacts of a common product base [28]. According to our definition, this is actually a strategy of increasing flexibility by introducing variability; the consequences are discussed below in section 7.

5. A static view on flexible development and product variability

This section argues that there is not such a thing as "good software engineering" in general. Software engineering, and more specifically, flexibility and variability have different meanings depending on the software product development model in use.

Most software intensive product builders consider software engineering as a broad domain that consists of a number of disciplines in which a large number of evolving software technologies can be applied (Figure 2). An exhaustive overview of the software engineering knowledge areas (disciplines) is provided in SWEBOK [18].

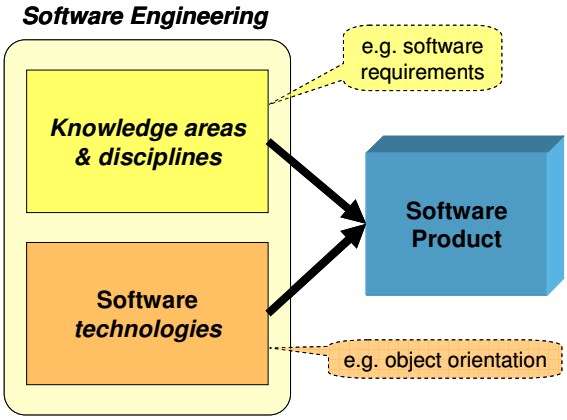


Figure 2. Software engineering as traditionally perceived by software intensive product builders

In practice, the various software engineering disciplines should be interpreted differently depending on the software product development model. For example, in the project based development model, requirements engineering boils down to gathering specific needs of a single customer (focus is on effective interviewing and understanding). In contrast, when making an out of the box product there is no direct contact with customers. Requirements engineering involves collaboration between different stakeholders to understand the domain (the focus is on decision making and conflict resolution). A similar analysis can be made for other disciplines of software engineering, such as architecture, testing, project management, etc.

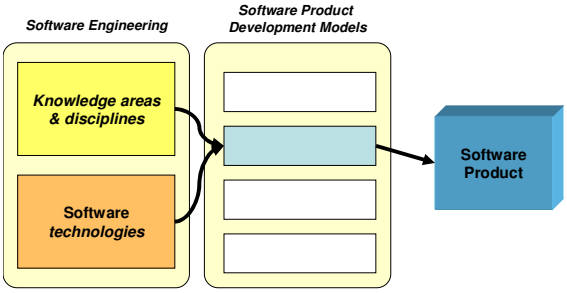


Figure 3. Software engineering through the glasses of the applied software product development model

Many companies consider "good software engineering" as a standalone domain, independent from the product development model. They neglect to interpret the software engineering disciplines in function of the used software product development model.

Understanding and dealing with the different interpretations of the software engineering disciplines in the context of a software product development model is challenging. Software intensive product builders should look at the domain of software engineering through the glasses of the product development model they apply (Figure 3). For software product lines, the Software Engineering Institute (SEI) defines a framework that lists the practice areas that need to be mastered in order to successfully create a software product line [34]. The practice areas explain how software engineering disciplines need to be interpreted for software product lines.

Furthermore, addressing the flexibility and variability challenges is not a responsibility of a single software engineering discipline. Usually it requires well aligned actions taken across several different disciplines at the same time. Flexibility and variability cross cut the disciplines. For example, introducing more variation points affects requirements engineering (one must understand what the variation points are), architecture (e.g. introduce a framework approach), but also the testing process (to deal with the combinatorial explosion of test scenarios that are the consequence of many variation points).

Because the interpretation of the software engineering disciplines depends on the applied software product development model, and the challenges of flexibility and variability cross-cut the software engineering disciplines, addressing flexibility and variability is different in the context of each software product development model.

6. A dynamic view on flexible development and product variability

Companies might be in a situation where they can afford to remain in the same software product development model for a longer period of time. Unfortunately, for many companies this is not the case. Evolutions in the business environment and the technological context can force a company to undergo a transition towards a different software product development model (see later in this section). For these companies choosing a flexibility and variability approach is not a one-off decision but a recurring one.

Companies are often not globally aware of an ongoing transition as it might be triggered by certain decisions that are taken by individuals or sub-departments of the organization (i.e. these transitions happen implicitly). Because of this (global) unawareness, companies neglect to re-interpret the software engineering disciplines in function of the new development model and as a consequence they also neglect to re-interpret their flexibility and variability approaches. This explains symptoms that companies are faced with such as "*software engineering practices that seemed to work well in the past do not seem to work anymore*" (e.g. customer interviewing techniques that worked well in a project based development approach are no longer efficient to define an out of the box product).

Deciding what the optimal model is and when to initiate a transition to that model is difficult but essential to remain effective and competitive. Several trade-offs need to be made, and sometimes remaining too long in the same model can be harmful.

In the remainder of this section an overview is given of the business and technological phenomena that can cause companies to transition (or to consider a transition) from one software product development model to another. These phenomena are occurring on a massive scale as almost every company is affected by at least one of them.

6.1. Increasing need for customer intimacy (variability)

It seems that for realizing customer intimacy the project based product development model is ideal. Each customer wish can be realized during the projects, without interference from other customers. Variability is offered by providing every customer with a separate product implementation. Applying the strategy of customer intimacy [23] can therefore cause transitions towards the project based development model (Figure

4). For example, companies developing out of the box products might move into other models that offer more customization opportunities.

6.2. Increasing need for bringing innovations faster to the markets (flexibility)

It seems that for realizing flexible development, the out of the box product development model is best suited. When possible, this is a very fast way of bringing solutions to customers: customers just buy the product and start using it.

In order to increase the speed of development [15], companies can consider transitioning towards the out of the box product development model (Figure 4). For example, a company doing project based development might consider moving to a model in which more domain engineering is involved so not every product has to be made from scratch.

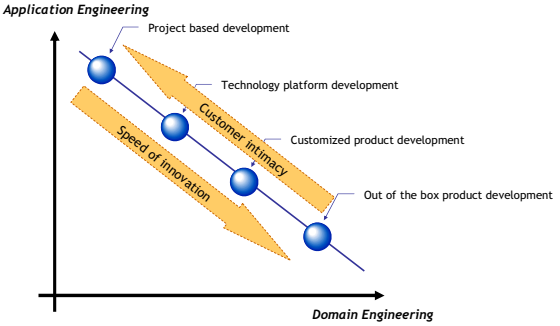


Figure 4. Transitions caused by an increase in customer intimacy and speed of innovation

6.3. Towards hybrid products: not too much product, not too much service

Cusumano observes [13] that many organizations are shifting to hybrid models that combine services with products. Many companies that have a 100% service offering today (e.g. consultancy) consider combining these services with a product offering (e.g. a tool). Companies that have a 100% product offering are attempting to complement it with services (e.g. as a reaction to decreasing product license revenue). This trend can result in transitions towards the customized product development model (Figure 5).

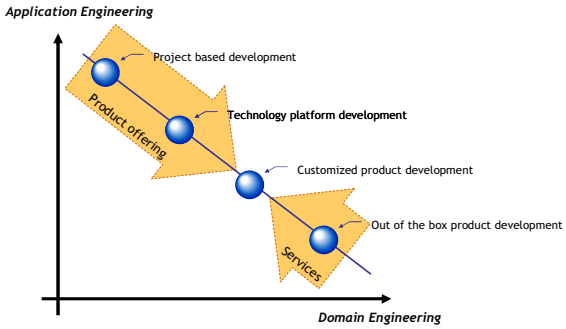


Figure 5. Transitions caused by combining products and services

6.4. Transitions induced by technological evolutions

Due to technological evolutions, products are no longer standalone, but become "platforms" used in other products/services (e.g. mashups, e.g. see [37]). By opening an API of a proprietary closed product, an entirely new range of combinations with other products becomes possible. This trend causes companies to transition to the platform model (Figure 6).

Other examples include transitions caused by the maturing process of technologies (Figure 6). Early adopter companies might not find mature COTS technology implementations, and, as a consequence, decide to build an (in-house) technology platform. Although this can be a perfectly valid strategy at the start, it can turn into a liability as soon as the state of the art becomes sufficiently mature. When this happens, companies might consider leaving the technology platform model and move to another model.

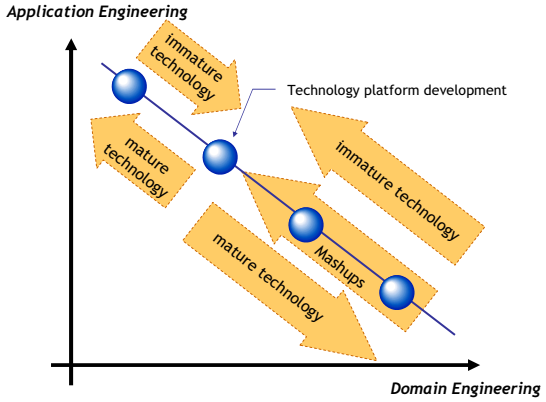


Figure 6. Transitions caused by technology maturation

7. A model for trading off flexibility and variability

Flexibility and variability have become major points of attention for many software intensive product builders. In the previous sections two perspectives have been analyzed. The static perspective reveals that flexibility and variability should be interpreted differently in each software product development model. The dynamic perspective reveals that if a change in the product strategy imposes a transition towards another software product development model, companies have to re-interpret their flexibility and variability approaches. This, of course, is only the case when speed of innovation or customer intimacy is relevant.

To increase the levels of product variability and flexibility, two strategies can be considered. The first one is to *consider moving to a different software product development model that better supports the flexibility and variability requirements*. Flexibility and variability seem to push companies in opposite directions (Figure 4). If both are important, a compromise has to be made (you can buy variability for flexibility and vice versa). In sections 7.1 and 7.2, the trade-offs that need to be made are discussed.

The second strategy is to *improve flexibility and/or variability within the model*. Sometimes the current development model is already the optimal one (taken the product strategy and other trade-offs into account). In that case, the only option is to improve flexibility and/or variability within the model. Discussion about this goes beyond the scope of this paper.

Figure 7 shows an overview of the proposed tradeoff model to improve flexibility and variability.

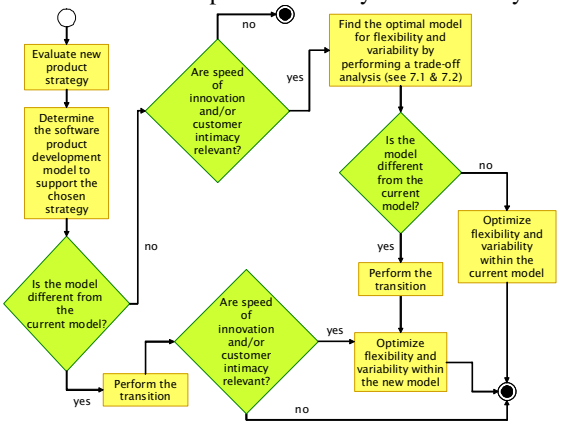


Figure 7. A model for trading off flexibility and variability

7.1. Trade-off analysis: choosing an optimal model for variability

First, let us consider product variability. As argued in 6.1 it seems that for realizing variability (a consequence of adhering to customer intimacy) the project based product development model is ideal. However, this ideal model is in many situations not feasible due to the following reasons.

7.1.1. Lack of expert resources. If the human resources required to engineer the product are highly skilled IT professionals, the organization simply might not have or cannot access to enough resources to apply this model. Studies [8] [17] have shown that innovation in software development is generated by a limited set of individuals: the critical resources. Furthermore, studies have shown that assigning engineers to more than two projects at the same time is wasteful [38]. The closer to the upper left corner in Figure 1, the more linear the relationship becomes between the number of developed products and the required expert resources to engineer them. The following trade-off criterion is therefore important: understanding to what extent one is prepared to pay the price for more variability in terms of assigning critical expert resources to individual customer projects.

7.1.2. Difficulty to realize mass customization. Development models in the upper left corner (Figure 1) require larger amount of customer-specific development activities (application engineering), meaning that less customers can be served with the same amount of resources. Companies are only able to manage a certain amount of many individual "customer threads" simultaneously. Understanding this threshold is important. Companies might become a victim of their own success (e.g. increased size of customer base) if they remain in models in the upper left corner. The following trade-off criterion is therefore important: understanding to what extent one is prepared to pay the price for more variability in terms of a smaller customer base[1].

7.1.3. High time-to-market pressure. The models situated in the upper left corner (Figure 1) usually require more time to develop products. In a situation with a high pressure to deliver fast, these models might not be optimal. The following trade-off criterion is

[1] An important consideration in adoption of models in the upper left corner is profitability. For some products and markets, customers are not ready to pay the premium price for customer specific developments that product builders need to be profitable.

therefore important: understanding to what extent one is prepared to pay the price for more variability in terms of less flexibility.

7.2. Trade-off analysis: choosing an optimal model for flexibility

Next, let us consider flexible development. It seems that for realizing flexibility, the out of the box product development model is best suited (section 6.2). However, this ideal model is in many situations not feasible due to the following reasons.

7.2.1. Lack of market and domain knowledge. Mastering successfully the models in the lower right corner (Figure 1) requires very good understanding of the domain (domain engineering) [11]. If this knowledge is not available, applying this model is risky because wrong assumptions can be made about customer needs. Companies can only increase flexibility by moving towards the lower right corner if this transition is accompanied by an acquisition of the relevant domain knowledge. The following trade-off criterion is therefore important: understanding to what extent one is prepared to pay the price for more flexibility in terms of increasing the investment involved in domain analysis.

7.2.2. Increased product complexity. Attempting to increase flexibility in the development process by adopting a product development model closer to the lower right corner (Figure 1), can have a (negative) side effect on the product. Increasing flexibility in this strategy is all about preparing and anticipating more flexibility in the future by introducing variation points. These variation points (often invisible to customers) are in the product architecture and add complexity to the product. For products with thousands of variation points (not an exceptional situation), the management of dependencies and interactions between these variation points might become difficult and may even result in degraded product quality. The following trade-off criterion is therefore important: understanding to what extent one is prepared to pay the price for more flexibility in terms of increasing product complexity. (Note that increasing flexibility by adopting an agile method does not suffer from the complexity problem. Only the process is affected, not the product.)

7.2.3. Difficulty of realizing variability. Models in the lower right corner (Figure 1) are typically characterized by a large customer base. In some situations defining a common product denominator for all these customers is difficult. Customers might require very different solutions (with even conflicting functionalities). Deriving products often requires much more than "simple" configuration. In this situation, both sophisticated product management activities (e.g. scoping [10]) and customization technologies are required (e.g. configuration languages, automatic code generation). The knowledge to master these technologies might not be available. Therefore, an important consideration is understanding to what extent one is prepared to pay the price for more flexibility in terms of less variability.

8. Conclusion

This paper addresses two fundamental decisions that many product builders are faced with: how to realize flexible development and how to realize product variability. Because these challenges are often mixed, companies sometimes fail to make the right decision. For example, a company might decide to develop an (internal) framework in order to speed up their development. However, if the company is not able to anticipate future wishes of its customers sufficiently well (e.g. due to the nature of the markets they are active in), the big investment in the framework might never be returned. In this situation a completely different strategy might be to introduce agile development methods. It is exactly the balancing between these two decisions with big impact that the presented tradeoff model is about.

Next research steps will extend this tradeoff model into a decision framework to guide companies in selecting and transitioning between flexibility and variability approaches.

9. Acknowledgments

The authors from Sirris would like to thank ISRIB (Institute for the encouragement of Scientific Research and Innovation of Brussels), IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders) and the Brussels-Capital Region for the support they provided for this research and publication. They would also like to thank their colleagues for the useful remarks provided on earlier versions of this document.

10. References

[1] Johnson, R. E. 1997. Frameworks = (Components + Patterns). *Communications of the ACM* 40(10):39-42.

[2] ACM. 2006b. Software Product Line. *Communications of the ACM* 49(12).
[3] Agile Alliance. http://www.agilealliance.org/.
[4] Bachmann, F., and L. Bass. 2001. Managing Variability in Software Architectures. SIGSOFT Software Engineering Notes 26(3):126-132.
[5] Beck, K., and C. Andres. 2004. *Extreme Programming Explained: Embrace Change*, 2nd Edition. Addison-Wesley.
[6] Beck, K., M. Beedle, A. van Bennekum, et al. 2001. Manifesto for Agile Software Development. http://www.agilemanifesto.org/.
[7] Bosch, J. 2002. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In *Proceedings of the Second Software Product Line Conference (SPLC2)*, Lecture Notes in Computer Science, volume 2379, 257-271. Springer-Verlag.
[8] Brooks, F. P. 1995. *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition, 2nd Edition. Addison-Wesley Professional.
[9] Christensen, C. M. 1997, *The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail*. Harvard Business School Press.
[10] Clements, P., and L. Northrop. 2002. *Software Product Lines: Practices and Patterns*. Addison-Wesley.
[11] Codenie, W., K. De Hondt, P. Steyaert, and A. Vercammen. 1997. From Custom Applications to Domain-Specific Frameworks. *Communications of the ACM* 40(10):70-77.
[12] Codenie, W., O. Biot, V. Blagojević, N. González-Deleito, T. Tourwé, and J. Deleu. 2008. Cruising the Bermuda Triangle of Software Development. Manuscript submitted for publication.
[13] Cusumano, M. A. 2004. *The Business of Software: What Every Manager, Programmer, and Entrepreneur Must Know to Thrive and Survive in Good Times and Bad*. Free Press.
[14] Czarnecki, K. 2005. Overview of Generative Software Development. In *Proceedings of Unconventional Programming Paradigms (UPP)*, Lecture Notes in Computer Science, volume 3566, 313-328. Springer-Verlag.
[15] Dehoff, K., and D. Neely. 2004. Innovation and Product Development: Clearing the New Performance Bar. Booz Allen Hamilton. http://www.boozallen.com/media/file/138077.pdf.
[16] Gamma, E., R. Helm, R. Johnson, and J. M. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
[17] Greenspun, P. 2002. Managing Software Engineers. http://philip.greenspun.com/ancient-history/managing-software-engineers.
[18] IEEE. 2004. Guide to the SWEBOK (Software Engineering Body of Knowledge). http://www.swebok.org/.
[19] ITEA. 2004. ITEA Technology Roadmap for Software-Intensive Systems, 2nd edition. http://www.itea-office.org/itea_roadmap_2.
[20] ITEA. 2005. ITEA Blue Book: European leadership in Software-intensive Systems and Services - The case for ITEA 2. http://www.itea2.org/itea_2_blue_book.
[21] Kang K., S. Cohen, J. Hess, W. Nowak, and S. Peterson. 1990. Feature-oriented domain analysis (FODA) feasibility study. Technical report CMU/SEI-90-TR-21, SEI CMU.
[22] Kiczales, G., J. Irwin, J. Lamping, J.-M. Loingtier, C. Videria Lopes, C. Maeda, and A. Mendhekar. 1996. Aspect-Oriented Programming. *ACM Computing Surveys* 28(4es):154.
[23] Kratochvil, M. 2005. *Growing Modular. Mass Customization of Complex Products, Services and Software*. Springer.
[24] Leadbeater, C. 2008. *We-think: Mass Innovation, not Mass Production*. Profile Books.
[25] Maiden, N., S. Robertson, and J. Robertson. 2006. Creative requirements: invention and its role in requirements engineering. In *Proceeding of the 28th International Conference on Software Engineering (ICSE'06)*, 1073-1074. ACM.
[26] Mernik, M., J. Heering, and A. M. Sloane. 2005. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys* 37(4):316-344.
[27] Nerur, S., R. Mahapatra, and G. Mangalaraj. 2005. Challenges of Migrating to Agile Methodologies. *Communications of the ACM* 48(5):72-78.
[28] Pohl, K., G. Böckle, and F. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer.
[29] Poppendieck, M., and T. Poppendieck. 2003. *Lean Software Development: An Agile Toolkit*. Addison-Wesley.
[30] Rubner, J. 2005. Tuned in to Today's Megatrends. In Siemens' *Pictures of the Future*, 90-91. http://www.siemens.com/Daten/siecom/HQ/CC/Internet/Research_Development/WORKAREA/fue_pof/templatedata/English/file/binary/PoF104art15_1321356.pdf.
[31] Schwaber, K. 2004. *Agile Project Management with Scrum*. Microsoft Press.
[32] Selic, B. 2003. The Pragmatics of Model-Driven Development. *IEEE Software* 20(5):19-25.
[33] Smith, P. G. 2007. *Flexible Product Development: Building Agility for Changing Markets*. Jossey-Bass.
[34] SEI SPL. Software Engineering Institute (SEI). Software product lines framework. http://www.sei.cmu.edu/productlines/framework.html.
[35] Treacy, M., and F. Wiersema. 1993. Customer Intimacy and Other Value Disciplines. *Harvard Business Review*.
[36] Ulrich, K. T., and S. D. Eppinger. 2000. *Product design and development*, 2nd Edition. McGraw-Hill. http://www.ulrich-eppinger.net/.
[37] Webmashup.com. The open directory of mashups and web 2.0 APIs, http://www.webmashup.com/.
[38] Wheelwright, S. C., and K. B. Clark. 1992. *Revolutionizing Product Development: Quantum Leaps in Speed, Efficiency, and Quality*. Free Press.

Deontic Logics for Modeling Behavioural Variability
*Research in progress**

P. Asirelli, M. H. ter Beek, S. Gnesi
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", ISTI–CNR
Via G. Moruzzi 1, I-56124 PISA (Italy)
{asirelli,terbeek,gnesi}@isti.cnr.it

A. Fantechi
DSI–Università di Firenze and ISTI–CNR, Pisa
Via S. Marta 3, I-50139 FIRENZE (Italy)
fantechi@dsi.unifi.it

Abstract

We discuss the application of deontic logics to the modeling of variabilities in product family descriptions. Deontic logics make it possible to express concepts like permission and obligation. As a first result of this line of research, we show how a Modal Transition System, a model that has recently been proposed as an expressive way to deal with behavioural variability in product families, can be completely characterized with deontic logic formulae. We moreover show some exemplary properties that can consequently be proved for product families. These preliminary results pave the way to a wider application of deontic logics to specifiy and verify variability in product families.

1 Introduction

A description of a Product Family (PF) is usually composed of a constant part and a variable part. The first part describes aspects that are common to all products of the family, whereas the second part represents those aspects, called *variabilities*, that will be used to differentiate one product from another. The modeling of variability has been extensively studied in the literature, especially that concerning *Feature modeling* [2, 6, 17]. In variability modeling the interest is in defining which features or components of a system are optional, alternative, or mandatory; techniques and tools are then developed to show that a product belongs to a family, or to derive instead a product from a family, by means of a proper selection of the features or components.

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

In this paper we are interested in the modeling of behavioural variability, that is, how the products of a family differ in their ability to respond to events in time: this is an aspect that the referenced techniques do not typically focus on. Many notations and description techniques have been recently proposed for this purpose, such as variants of UML state diagrams [3, 25] or variants of UML sequence diagrams, for example STAIRS [20]; another proposal can be found in [24], where UML state diagrams and sequence diagrams have been enriched with aside notations to describe variation points. At this regard, we rather prefer to look for an expressive modeling formalism for families based on the choice of a basic behavioural model, namely Labelled Transition Systems (LTSs), which is one of the most popular formal frameworks for modeling and reasoning about the behaviour of a system. Modal Transition Systems (MTSs) have been proposed, in several variants, to model a family of such LTSs [18, 13, 8]: in this way it is possible to embed in a single model the behaviour of a family of products that share the basic structure of states and transitions, transitions which however can be seen as mandatory or possible for the products of the family.

Deontic logics [1] have become very popular in computer science in the last few decades to formalize descriptional and behavioural aspects of systems. This is mainly because they provide a natural way to formalize concepts like violation, obligation, permission, and prohibition. Intuitively, they permit one to distinguish between correct (normative) states and actions on the one hand and non-compliant states and actions on the other hand. This makes deontic logics a natural candidate for expressing the variability of a family of products. Recently, a Propositional Deontic Logic (PDL) capable of expressing the permitted

behaviour of a system has been proposed [5]. We want to study in detail the application of this kind of logics to the modeling of behavioural variability. Indeed, PDL appears to be a good candidate to express in a unique framework both behavioural aspects, by means of standard branching-time logic operators, and constraints over the product of a family, which usually require a separate expression in a first-order logic (as seen in [2, 9, 21]).

In this paper, we want to focus our attention on the ability of a logic of finitely characterizing the complete behaviour of a system, that is, given a MTS $\mathcal{A}$, we look for a formula of the logic that is satisfied by all and only those Labelled Transition Systems that can be derived from $\mathcal{A}$ As a first result in this direction, the main contribution of this paper is that we are able to finitely characterize finite state MTSs, using a deontic logic; to this aim, we associate a logical formula (called *characteristic formula* [11, 12, 23]) to each state of the MTS. Consequently, every LTS of the family defined by the MTS satisfies the formula, and no LTS outside the family satisfies it. In this way we establish a link between a common model of behavioural variability and PDL. The deontic logic proposed in this paper is able to describe, in a faithful way, the behaviour of systems modelled by finite state MTSs. Our work can serve as a basis to develop a full logical framework to express behavioural variability and to build verification tools employing efficient model-checking algorithms.

2 Labelled Transition Systems

As said before, a basic element of our research is the concept of a Labelled Transition System, of which we define several variants.

Definition 2.1 (Labelled Transition System) *A* Labelled Transition System *(LTS) is a quadruple* $(Q, q_0, Act, \rightarrow)$*, in which*

- Q *is a set of states;*
- $q_0 \in Q$ *is the initial state;*
- *Act is a finite set of observable events (actions);*
- $\rightarrow \subseteq Q \times Act \times Q$ *is the transition relation; instead of* $(q, \alpha, q') \in \rightarrow$ *we will often write* $q \xrightarrow{\alpha} q'$*.*

Definition 2.2 (Modal Transition System) *A* Modal Transition System *(MTS) is a quintuple* $(S, s_0, Act, \rightarrow_\Box, \rightarrow_\Diamond)$ *such that* $(S, s_0, Act, \rightarrow_\Box \cup \rightarrow_\Diamond)$ *is a LTS. A MTS has two distinct transition relations: the* must *transition relation* $\rightarrow_\Box$ *expresses* required *transitions, while the* may *transition relation* $\rightarrow_\Diamond$ *expresses* possible *transitions.*

A MTS defines a family of LTSs, in the sense that each LTS $P = (S_P, p_0, Act, \rightarrow)$ of the family can be obtained from the MTS $F = (S_F, f_0, Act, \rightarrow_\Box, \rightarrow_\Diamond)$ by considering its transition relation $\rightarrow$ to be $\rightarrow_\Box \cup R$, with $R \subset \rightarrow_\Diamond$, and pruning the states that are not reachable from its initial state p_0. The "P *is a product of* F" relation below, also called "conformance" relation, links a MTS F representing a family with a LTS P representing a product.

Definition 2.3 (Conformance relation) *We say that* P is a product of F*, denoted by* $P \vdash F$*, if and only if* $p_0 \vdash s_0$*, where* $p \vdash f$ *if and only if*

- $f \xrightarrow{a}_\Box f' \implies \exists p' \in S_P : p \xrightarrow{a} p'$ *and* $p' \vdash f'$
- $p \xrightarrow{a} p' \implies \exists f' \in S_F : f \xrightarrow{a}_\Diamond f'$ *and* $p' \vdash f'$

Another extension of LTSs is obtained by labelling its states with atomic propositions, leading to the concept of doubly-labelled transition systems [7].

Definition 2.4 (Doubly-Labelled Transition System) *A* Doubly-Labelled Transition System *(L^2TS) is a quintuple* $(Q, q_0, Act, \rightarrow, AP, L)$*, in which*

- $(Q, q_0, Act, \rightarrow)$ *is a LTS;*
- *AP is a set of atomic propositions;*
- $L : Q \longrightarrow 2^{AP}$ *is a labelling function that associates a subset of AP to each state of the LTS.*

3 A deontic logic

Deontic logics are an active field of research in formal logic for many years now. Many different deontic logic systems have been developed and in particular the use of modal systems has had a lot of success in the deontic community [1]. The way such logics formalize concepts such as violation, obligation, permission and prohibition is very useful for system specification, where these concepts arise naturally. In particular, deontic logics seem to be very useful to formalize product families specifications, since they allow one to capture the notion of possible and compulsory features.

Our starting point is the Propositional Deontic Logic (PDL) defined in [5]. PDL is able to express both the evolution in time of a system by means of an action, and the fact that certain actions are permitted or not in a given state. The original definition considers actions from a set *Act*, each action producing a set of events from a set E. The set of events produced by an action $\alpha \in Act$ is named $I(\alpha)$. The logic we propose in this paper is a temporal extension of PDL,

in a style reminiscent of the extension proposed in [5]. The syntax of the logic is:

$$\phi ::= tt \mid p \mid \neg\phi \mid \phi \wedge \phi' \mid A\pi \mid E\pi \mid [\alpha]\phi \mid P(\alpha) \mid P_w(\alpha)$$
$$\pi ::= \phi \; U \; \phi'$$

As usual, *ff* abbreviates $\neg tt$, $\phi \vee \phi'$ abbreviates $\neg(\neg\phi \wedge \neg\phi')$, and $\phi \implies \phi'$ abbreviates $\neg\phi \vee \phi'$. Moreover, $EF\phi$ abbreviates $E\,(tt\; U\; \phi)$ and $AG\phi$ abbreviates $\neg EF\neg\phi$. Finally, the informal meaning of the three non-conventional modalities, explained below in more detail, is:

- $[\alpha]\phi$: after any possible execution of α, ϕ holds;
- $P(\alpha)$: every way of executing α is allowed;
- $P_w(\alpha)$: some way of executing α is allowed.

The first of these three modalities thus provides the possibility to express evolution, while the other two provide the possibility to express (weak) permission. Furthermore, $\langle\alpha\rangle\phi$ abbreviates $\neg[\alpha]\neg\phi$.

The two variants of permission are rather common in the literature on deontic logic. The operator $P(\alpha)$ tells whether or not an action α is allowed to be performed. It can be called a strong permission since it requires that every way of performing α has to be allowed (e.g. if we were to say that *driving* is allowed, it would mean that also *driving while drinking beer* is allowed). Not surprisingly, permission has been a polemical notion since the very beginning of deontic logic. Some have proposed a weak version [22] in which to be allowed to perform an action means that this action is allowed only in some contexts. We stick to [5] and use both notions of permission. The latter is denoted by the operator $P_w(\alpha)$, which must be read as α is weakly allowed. The two versions differ in their properties (see [5] for details).

In [5] both variants of permission are used to define obligation $O(\alpha)$ as $P(\alpha) \wedge \neg P_w(\neg\alpha)$, i.e. α is obligated if and only if it is strongly permitted and no other action is weakly allowed. This definition avoids Ross's paradox $O(\alpha) \implies O(\alpha \vee \alpha')$, which can be read as "if you are obliged to send a letter, then you are obliged to send it or burn it".

The formal semantics of our logic is given below by means of an interpretation over L^{2}TS, mimicking the original semantics of PDL in [5]. To this aim, the L^{2}TS used as an interpretation structure is defined as a sixtuple $(W, w_0, E, \rightarrow, AP \cup E, L)$, in which the transitions are labelled over the set of events E and the states (corresponding to the *worlds* of the standard interpretation) are labelled with atomic propositions as well as with the events allowed in the states. To this purpose, we also use a relation $P \subseteq W \times E$ to denote which events are permitted in which world, with the understanding that $P(w, e)$ if and only if $e \in L(w)$.

Definition 3.1 (Semantics) *The satisfaction relation of our deontic logic is defined as follows:*

- $w \models tt$ *always holds;*
- $w \models p$ *iff* $p \in L(w)$*;*
- $w \models \neg\phi$ *iff not* $w \models \phi$*;*
- $w \models \phi \wedge \phi'$ *iff* $w \models \phi$ *and* $w \models \phi'$*;*
- $w \models A\pi$ *iff* $\sigma \models \pi$ *for all paths* σ *that start with state* w*;*
- $w \models E\pi$ *iff there exists a path* σ *that starts with state* w *such that* $\sigma \models \pi$*;*
- $w \models [\alpha]\phi$ *iff* $\forall e \in \mathcal{I}(\alpha) : w \xrightarrow{e} w'$ *implies* $w' \models \phi$*;*
- $w \models P(\alpha)$ *iff* $\forall e \in \mathcal{I}(\alpha) : P(w, e)$ *holds;*
- $w \models P_w(\alpha)$ *iff* $\exists e \in \mathcal{I}(\alpha) : P(w, e)$ *holds;*
- $\sigma \models [\phi \; U \; \phi']$ *iff there exists a state* s_j*, for some* $j \geq 0$*, on the path* σ *such that for all states* s_k*, with* $j \leq k$*,* $s_k \models \phi'$ *while for all states* s_i*, with* $0 \leq i < j$*,* $s_i \models \phi$*.*

4 A deontic characteristic formula for MTSs

In this section, we show how a unique deontic logic formula can completely characterize a family of LTSs by separating the structure of the LTS (taken care of by the box fomulae) from the optional/mandatory nature of the transitions (taken care of by the permission formulae). Since a MTS is a compact expression of a family of LTSs, this is equivalent to saying that we are able to characterize a MTS with a deontic logic formula. The result we show here is rather preliminary, in the sense that it currently needs the following simplifying assumptions, but it nevertheless shows the potentiality of our deontic logic.

- First, we adopt a strict interpretation to the permitted events that label the transitions of a L^{2}TS, namely we assume that $w \xrightarrow{e}$ implies $P(w, e)$, that is, only permitted actions are executed.
- We then assume, for any action α, that $I(\alpha) = \{e_\alpha\}$, that is, actions and events are indistinguishable.
- We also assume that a MTS defines the family of those LTSs that are derived from a corresponding family of L^{2}TSs, simply ignoring the predicates on the states.
- Last, we use a simpler form of MTSs, in which transitions leaving the same state are either all box transitions or all diamond transitions. As we show next, this assumption allows us to distinguish box states and diamond states, and have a single transition relation.

Definition 4.1 (Alternative def. MTS) *A MTS is a quintuple* $(BS, DS, s_0, Act, \rightarrow)$ *such that* $(BS \cup DS, s_0, Act, \rightarrow)$ *is a LTS and* $BS \cap DS = \varnothing$. *A MTS has two distinct sets of states: the* box *states* BS *and the* diamond *states* DS.

At this point, we define the characteristic formula $\mathcal{FC}(M)$ of a (simple) MTS $M = (BS, DS, s_0, Act, \rightarrow)$ as $\mathcal{FC}(s_0)$, where

$$\mathcal{FC}(s) = \begin{cases} (\bigvee_i P_w(\alpha_i)) \wedge ((\bigwedge_i [\alpha_i]\mathcal{FC}(s_i)) & \text{if } s \in DS \\ (\bigwedge_i O(\alpha_i)) \wedge ((\bigwedge_i [\alpha_i]\mathcal{FC}(s_i)) & \text{if } s \in BS \\ \text{and } \forall i \colon s \xrightarrow{e_i} s_i \text{ with } I(\alpha_i) = \{e_i\} \end{cases}$$

If we define the characteristic formula in an equational form using the expressions above, we obtain one equation for each state of the MTS, and the equations have a number of terms equal to two times the number of transitions leaving the relevant state. An attempt to write a single characteristic formula gives a formula exponential in size with respect to the number of states, and needs some form of fixed point expression for expressing cycles in the MTS (see [12]).

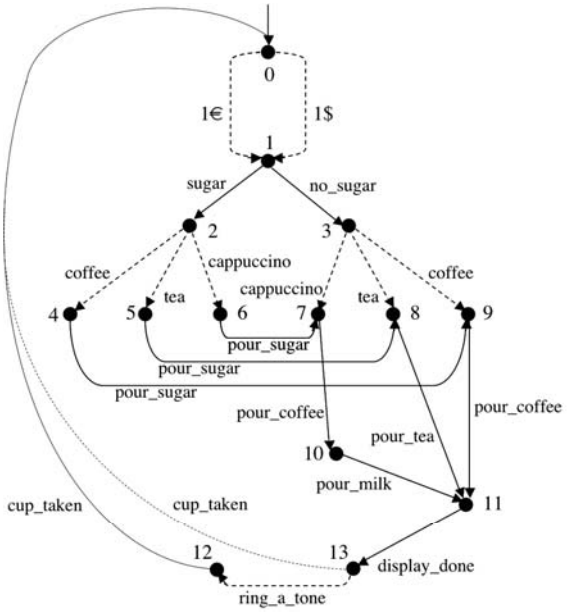


Figure 1. A MTS modeling a product family.

5 An example

Let us consider the example introduced in [8], that is, a family of coffee machines represented by the MTS depicted in Fig. 1, which allows products to differ for the two different currencies accepted, for the three drinks delivered and for the presence of a ring tone after delivery. In the figure, solid arcs are *required* transitions and dashed arcs are *possible* transitions, that is, states with outgoing solid arcs belong to BS, and states with outgoing dashed arcs belong to DS.

The characteristic formula in equational form is given by the following set of equations:

$$\begin{aligned}
\phi_0 &= (P_w(1€) \vee P_w(1\$)) \wedge ([1€]\phi_1 \wedge [1\$]\phi_1) \\
\phi_1 &= (O(\text{sugar}) \wedge O(\text{no_sugar})) \\
&\quad \wedge ([\text{sugar}]\phi_2 \wedge [\text{no_sugar}]\phi_3) \\
\phi_2 &= (P_w(\text{coffee}) \vee P_w(\text{cappuccino}) \vee P_w(\text{tea})) \\
&\quad \wedge ([\text{coffee}]\phi_4 \wedge [\text{cappuccino}]\phi_5 \wedge [\text{tea}]\phi_6) \\
\phi_3 &= (P_w(\text{coffee}) \vee P_w(\text{cappuccino}) \vee P_w(\text{tea})) \\
&\quad \wedge ([\text{coffee}]\phi_7 \wedge [\text{cappuccino}]\phi_8 \wedge [\text{tea}]\phi_9) \\
\phi_4 &= O(\text{pour_sugar}) \wedge [\text{pour_sugar}]\phi_7 \\
\phi_5 &= O(\text{pour_sugar}) \wedge [\text{pour_sugar}]\phi_8 \\
\phi_6 &= O(\text{pour_sugar}) \wedge [\text{pour_sugar}]\phi_9 \\
\phi_7 &= O(\text{pour_coffee}) \wedge [\text{pour_coffee}]\phi_{10} \\
\phi_8 &= O(\text{pour_tea}) \wedge [\text{pour_tea}]\phi_{11} \\
\phi_9 &= O(\text{pour_coffee}) \wedge [\text{pour_coffee}]\phi_{11} \\
\phi_{10} &= O(\text{pour_milk}) \wedge [\text{pour_milk}]\phi_{11} \\
\phi_{11} &= O(\text{display_done}) \wedge [\text{display_done}]\phi_{12} \\
\phi_{12} &= (P_w(\text{cup_taken}) \vee P_w(\text{ring_a_tone})) \\
&\quad \wedge ([\text{cup_taken}]\phi_0 \wedge [\text{ring_a_tone}]\phi_{13}) \\
\phi_{13} &= O(\text{cup_taken}) \wedge [\text{cup_taken}]\phi_0
\end{aligned}$$

Note that the characteristic formula given above does not allow, from any state of the considered MTS, to derive a LTS such that the corresponding state has no outgoing transitions, even in the case of diamond states.

The characteristic formula of a MTS implies any other property which is satisfied by the MTS, and can thus serve as a basis for the logical verification over MTSs. Actually, this approach is not as efficient as model-checking ones, but the definition of the characteristic formula may serve as a basis for a deeper study of the application of deontic logics to the verification of properties of families of products.

We now show two exemplary formulae that use deontic operators to formalize properties of the products derived by the family of coffee machines represented by the MTS depicted in Fig. 1.

1. The family permits to derive a product in which it is permitted to get a coffee with 1€:

$$P_w(1€) \implies [1€]\, E\, (tt\, U\, P_w(\text{coffee}))$$

2. The family obliges every product to provide the possibility to ask for sugar:

$$A\, (tt\, U\, O(\text{sugar}))$$

Notice that the deontic operators predicate on the relations between the products and the family, although they are defined on particular states of their behaviour.

It can be seen that the formulae above can be derived, using the axiom system given in [5], from the characteristic formula of the MTS of Fig. 1. This proves that the above properties are actually verified for the coffee machines that belong to the family described by the MTS.

6 Conclusions

We have shown how deontic logics can express the variability of a family, in particular by showing the capability of a deontic logic formula to finitely characterize a finite state Modal Transition System, a formalism proposed to capture the behavioural variability of a family. The logical framework has allowed us to prove simple behavioural properties of the example MTS shown.

These results, although very preliminary, lay the basis of further research in this direction, in which we aim at exploiting the full power of PDL for what concerns the expression of behavioural variability: the presence of CTL-like temporal operators allows more complex behavioural properties to be defined and therefore more expressive descriptions of behavioural variability to be supported. In particular, the dependency between variation points could be addressed in a uniform setting. Another interesting direction is the adoption of model-checking techniques to build efficient verification tools aimed at verifying, on the family definition, properties which are inherited by all the products of the family.

It would also be interesting to have a look at the variability of dynamic models in the large, taking into account both the problem of modelling variability in business process models and that of using goal modelling—which intrinsically includes variability—to model variable processes.

Finally, it remains to study to what degree the complexity of the proposed logic and verification framework can be hidden from the end user, or be made more user friendly, in order to support developers in practice.

References

[1] L. Åqvist, Deontic Logic. In D. Gabbay and F. Guenthner (Eds.): Handbook of Philosophical Logic (2nd Edition), Volume 8. Kluwer Academic, Dordrecht, 2002, 147-264.

[2] D.S. Batory, Feature Models, Grammars, and Propositional Formulas. In J.H. Obbink and K. Pohl (Eds.): *Proceedings Software Product Lines Conference (SPLC'05)*, LNCS 3714, 2005, 7–20.

[3] J. Bayer, S. Gerard, O. Haugen, J. Mansell, B. Møller-Pedersen, J. Oldevik, P. Tessier, J.-P. Thibault and T. Widen, Consolidated Product Line Variability Modeling. Chapter 6 of [16], 2006, 195–241.

[4] M.H. ter Beek, A. Fantechi, S. Gnesi and F. Mazzanti, An action/state-based model-checking approach for the analysis of communication protocols for Service-Oriented Applications. In S. Leue and P. Merino (Eds.): *Proceedings Formal Methods for Industrial Critical Systems (FMICS'07)*, LNCS 4916, Springer, 2008, 133–148.

[5] P.F. Castro and T.S.E. Maibaum, A Complete and Compact Propositional Deontic Logic. In C.B. Jones, Zh. Liu and J. Woodcock (Eds.): *International Colloquium Theoretical Aspects of Computing (ICTAC'07)*, LNCS 4711, Springer, 2007, 109–123.

[6] K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, Boston, MA, 2000.

[7] R. De Nicola and F.W. Vaandrager, Three Logics for Branching Bisimulation. *Journal of the ACM* 42, 2 (1995), 458–487.

[8] A. Fantechi and S. Gnesi, Formal Modeling for Product Families Engineering. In *Proceedings Software Product Lines Conference (SPLC'08)*, IEEE, 2008, 193–202.

[9] A. Fantechi, S. Gnesi, G. Lami and E. Nesti, A Methodology for the Derivation and Verification of Use Cases for Product Lines. In R.L. Nord (Ed.): *Proceedings Software Product Lines Conference (SPLC'04)*, LNCS 3154, Springer, 2004, 255–265.

[10] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese and F. Tiezzi, A model checking approach for verifying COWS specifications. In J.L. Fiadeiro and P. Inverardi (Eds.): *Proceedings Fundamental Approaches to Software Engineering (FASE'08)*, LNCS 4961, Springer, 2008, 230–245.

[11] A. Fantechi, S. Gnesi and G. Ristori, Compositionality and Bisimulation: A Negative Result. *Information Processing Letters* 39, 2 (1991), 109–114.

[12] A. Fantechi, S. Gnesi and G. Ristori, Modeling Transition Systems within an Action Based Logic. Technical Report B4-49-12, IEI–CNR, Dec. 1995.

[13] D. Fischbein, S. Uchitel and V.A. Braberman, A Foundation for Behavioural Conformance in Software Product Line Architectures. In R.M. Hierons and H.

Muccini (Eds.): *Proceedings Role of Software Architecture for Testing and Analysis (ROSATEA'06)*, ACM, 2006, 39–48.

[14] A. Gruler, M. Leucker and K.D. Scheidemann, Modeling and Model Checking Software Product Lines. In G. Barthe and F.S. de Boer (Eds.): *Proceedings Formal Methods for Open Object-Based Distributed Systems (FMOODS'08)*, LNCS 5051, Springer, 2008, 113–131.

[15] G. Halmans and K. Pohl, Communicating the Variability of a Software-Product Family to Customers, *Software and Systems Modeling* 2, 1 (2003), 15–36.

[16] T. Käkölä and J.C. Dueñas (Eds.): *Software Product Lines—Research Issues in Engineering and Management*, Springer, Berlin, 2006.

[17] K. Kang, S. Choen, J. Hess, W. Novak and S. Peterson, Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical Report SEI-90-TR-21, Carnegie Mellon University, Nov. 1990.

[18] K.G. Larsen, U. Nyman and A. Wąsowski, Modal I/O Automata for Interface and Product Line Theories. In R. De Nicola (Ed.): *Proceedings European Symposium on Programming Languages and Systems (ESOP'07)*, LNCS 4421, Springer, 2007, 64–79.

[19] K.G. Larsen, U. Nyman and A. Wąsowski, Modeling software product lines using color-blind transition systems. *International Journal on Software Tools for Technology Transfer* 9, 5–6 (2007), 471–487.

[20] M.S. Lund and K. Stølen, A Fully General Operational Semantics for UML 2.0 Sequence Diagrams with Potential and Mandatory Choice. In J. Misra, T. Nipkow and E. Sekerinski (Eds.): *Proceedings Formal Methods (FM'06)*, LNCS 4085, Springer, 2006, 380–395.

[21] M. Mannion and J. Camara, Theorem Proving for Product Line Model Verification. In F. van der Linden (Ed.): *Proceedings Software Product-Family Engineering (PFE'03)*, LNCS 3014, Springer, 2004, 211–224.

[22] J.-J.Ch. Meyer, A Different Approach to Deontic Logic: Deontic Logic Viewed as a Variant of Dynamic Logic. *Notre Dame Journal of Formal Logic* 29, 1 (1988), 109–136.

[23] B. Steffen, Characteristic Formulae. In G. Ausiello, M. Dezani-Ciancaglini and S. Ronchi Della Rocca (Eds.): *Proceedings International Colloquium Automata, Languages and Programming (ICALP'89)*, LNCS 372, Springer, 1989, 723–732.

[24] K. Pohl, G. Böckle and F. van der Linden, *Software Product Line Engineering—Foundations, Principles, and Techniques*, Springer, Berlin, 2005.

[25] T. Ziadi and J.-M. Jézéquel, Product Line Engineering with the UML: Deriving Products. Chapter 15 of [16], 2006, 557–586.

Structuring the Product Line Modeling Space: Strategies and Examples

Paul Grünbacher Rick Rabiser Deepak Dhungana
Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler Universität Linz, Austria
gruenbacher@ase.jku.at

Martin Lehofer
Siemens VAI, Turmstrasse 44
Linz, Austria
martin.lehofer.ext@siemens.com

Abstract

The scale and complexity of real-world product lines makes it practically infeasible to develop a single model of the entire system, regardless of the languages or notations used. Product line engineers need to apply different strategies for structuring the modeling space to facilitate the creation and maintenance of the models. The combined use of these strategies needs to support modeling variability at different levels of abstraction. In this paper, we discuss some of these strategies and provide examples of applying them to a real-world product line. We also describe tool support for structuring the modeling space and outline open research issues.

1. Introduction and Motivation

Many software product lines today are developed and maintained using a model-based approach. Numerous approaches are available for defining product lines such as feature-oriented modeling languages [19, 5, 1], decision-based approaches [7, 26], orthogonal approaches [3], architecture modeling languages [21, 6], or UML-based techniques [2, 11]. Numerous tools have been developed to automate domain and application engineering activities based on these models.

No matter which modeling approach is followed, developing a single model of a product line is practically infeasible due to the size and complexity of today's systems. The high number of features and components in real-world systems means that modelers need strategies and mechanisms to organize the modeling space. *Divide and conquer* is a useful principle but the question remains which concrete strategies can be applied to divide and structure the modeling space.

The aspect of organizing modeling spaces in product line engineering has already been addressed by some researchers: For instance, Hunt [16] discusses the challenge of organizing an asset base for product derivation. Jorgenson [18] presents an approach to model a product line at different levels of abstraction. In our own research we have proposed an approach that addresses the more technical aspect of creating and merging multiple model fragments [8]. While these approaches give some initial answers, structuring the modeling space remains challenging. Due to the absence of guidelines product line engineers still struggle with this task. It is difficult if not impossible to define general recommendations regarding the optimal approach for a certain development context. For instance, modelers can structure models by using problem space structures (e.g., different markets) or solution space structures (e.g., different subsystems). Alternatively, they can use a combination of approaches. The problem of structuring the modeling space also exists when engineering single software systems. However, we have applied and tested our approach in the context of product line variability modeling, where it is necessary to understand and define the variability of the system at different levels of abstraction. Modelers lack guidelines on how to treat variability aspects in a large modeling spaces. E.g., variability might be defined as part of existing models or in separate models.

This paper is not about notations and modeling languages for product lines. Instead, we explore the issue of how to structure models for large product lines regardless of the modeling language used. We discuss

several strategies for structuring product line models from different perspectives (e.g., solution space, organization, business needs, ...). We present our own experiences of using a multi-level structuring approach. We briefly discuss necessary tool capabilities and point out open issues to be addressed in future research.

2. Structuring the Modeling Space

There are many ways for modeling and managing variability but the basic challenges remain: Product line engineers need to define the variability of the *problem space*, i.e., stakeholder needs and desired features; the variability of the *solution space*, i.e., the architecture and the components of the technical solution; and the dependencies between these two.

Regardless of the concrete modeling approach used there are several options to structure and organize the modeling space:

Mirroring the Solution Space Structure. Whenever product lines are modeled for already existing software systems, the structure of the available reusable assets can provide a basis for organizing the modeling space. Models can be created that reflect the structure of the technical solution. This can be done by creating separate variability models for different subsystems of a product line. For example, the package structure of a software system or an existing architecture description can serve as a starting point. The number of different models should be kept small to avoid negative effects on maintainability and consistency. This strategy can be suitable for instance if the responsibilities of developers and architects for certain subsystems are clearly established.

Decomposing into Multiple Product Lines. On a larger scale complex products are often organized using a multi-product line structure [25]. For example, complex software-intensive system such as cars or industrial plants with *system of systems* architectures may contain several smaller product lines as part of the larger system. Models have to be defined for each of these product lines and kept consistent during domain and application engineering. This strategy often means that different teams create variability models for the product line they are responsible for.

Structuring by Asset Type. Another way of dealing with the scale of product line models is to structure the modeling space based on the asset types in the domain. Separate models can then be created for different types of product line assets. Examples are requirements variability models based on use cases [13], architecture variability models [6], or documentation variability models for technical and user-specific documents [17]. This approach is in line with orthogonal approaches [23] that suggest using few variability models that are related with possibly many asset models. Structuring by asset type allows managing variability in a coherent manner. It is however important to consider the dependencies between the different types of artifacts which can cause additional complexity.

Following the Organizational Structure. This strategy suggests to follow the structure of the organization when creating product line models. Different stakeholders are interested in different concerns of a product line [10]. In many organizations architectural knowledge is distributed across different stakeholders independent of their roles and responsibilities in the development process. Conway's Law [4] states that "... organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations". In a multi-team environment, individual teams collaborate closely on certain aspects of a product line. It can thus be a good strategy to structure the product line modeling space based on the team structure to reflect the modeling concerns of the involved stakeholder groups. However, creating product line models driven by stakeholders can easily increase the redundancy in models. While such an approach is already useful in single system engineering it might bring additional benefits when migrating towards a product line approach: The knowledge about variability is typically not explicitly available in such a situation. Understanding and defining the variability at the level of a small team might be easier to accomplish than following a top-down approach.

Considering Cross-cutting Concerns. Using concepts from aspect-oriented development to structure product line models is helpful when many crosscutting features need to be described. Aspect-oriented product line modeling can be used to model both problem and solution space variability. For instance, Völter and Groher [27] describe an approach that involves creating a model of the core features all products have in common and defining aspect variability models for product-specific features shared by only some products. Complex aspect dependencies can however lead to difficulties of managing their interaction.

Focusing on Market Needs. Structuring the modeling space can also be driven by business and management considerations, e.g., from product management [15] or marketing [20]. Focusing variability modeling on business considerations eases the communication with customers. If combined with other strategies this approach can support the communication between customers and sales people. If following this strategy in pure form, models might be unrelated with the tech-

nical solution thus leading to problems when trying to understand the actual realization of the variability.

3. Applying the Strategies: Two Examples

In typical product lines it does not make sense to use the described strategies in their pure form. Instead in practical settings the different approaches have to be combined: For example, a system of systems can be modeled for different customer types while at the same time also structuring the resulting models by asset types. An example is the domain of mobile phones, where models can be created for different product lines for senior citizens, teenagers, business people. In each of these product lines one can define models for the various assets (requirements, architecture, and documentation). Another example is to first create models driven by product and marketing considerations and later restructure these models to follow the solution space structure. The importance of hybrid approaches is also reflected in recent research. For instance, several papers appeared about linking problem space and solution space models [7, 14, 22].

Over the last three years we have been collaborating with Siemens VAI, the world's leading engineering and plant building company for the iron, steel, and aluminum industries. In the context of the research project we have developed the decision-oriented product line engineering approach DOPLER [7] and supporting tools [9]. A main aim of the research project has been to validate our tool-supported approach by creating product line models for our industry partner's process automation system for continuous casting in steel plants (CL2). Our initial approach of putting all model elements into one model failed due to the scale and complexity of the product line. We therefore experimented with different strategies as described in the previous section. We did not explicitly choose a single strategy but started to combine them.

3.1. Initial Approach: Solution Space Structure and Business Concerns

Our initial strategy was to use the existing technical solution structure as the starting point for structuring our models. Due to the size of the CL2 product line, creating one model representing the whole technical solution was infeasible due to the distributed knowledge about variability and the overall complexity of the system. Also, such a strategy only inadequately supported evolution in the multi-team development environment of Siemens VAI [8]. In the company's development process different teams are in charge for various

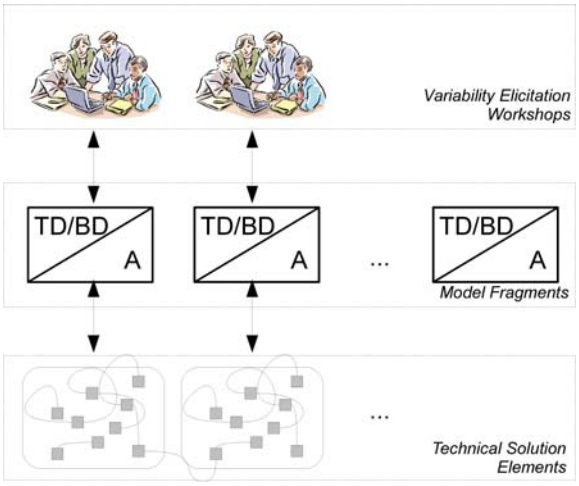


Figure 1. The initial strategy considers the solution space structure and business concerns. Variability identified in workshops and with automated tools is captured in different model fragments reflecting different subsystems. Model fragments contain technical decisions (TD), business decisions (BD), and reusable assets (A).

subsystems of the product line (e.g., cooling, cutting, material tracking, etc.). People in charge of a particular subsystem have intimate knowledge about this part of the product line, however, no single developer has the knowledge required to model the entire system. We therefore came up with a multi-team approach based on the idea of variability model fragments [8].

The variability of the system was elicited in two ways (see Figure 1): (i) We conducted *moderated workshops* with engineers and project managers representing various subsystems to identify major differences of products delivered to customers and to formally describe these differences in models [24]. Business decisions (BD) represent external variability. Technical decisions (TD) define internal variability not visible to customers. (ii) We used *automated tools* to analyze the technical variability at the level of components by parsing existing configuration files. Developers had to manually confirm the variation points detected by our tools. We created variability model fragments for 11 subsystems of CL2. The developed variability model fragments vary in size and complexity due to the different scope of the subsystems. The average model fragment contains 50 assets (mainly components and parameters), 12 decisions and 23 references to other model fragments [8].

The approach has several advantages: It supports

the clear *separation of concerns* through variability model fragments for different subsystems. This eases model development and evolution in multi-team modeling environments. Different *stakeholder perspectives* are considered by modeling both business and technical variability of the system. However, we noticed two weaknesses which made it necessary to further refine the approach and to revisit the structure of the modeling space: *Mixing technical and business decisions* in one model fragment can cause problems as different people are responsible for maintaining these decisions. *Mixing assets and decisions* in one model fragment has negative effects on model maintenance. While assets change often, decisions turned out to be more stable. We thus decided to refine our initial approach.

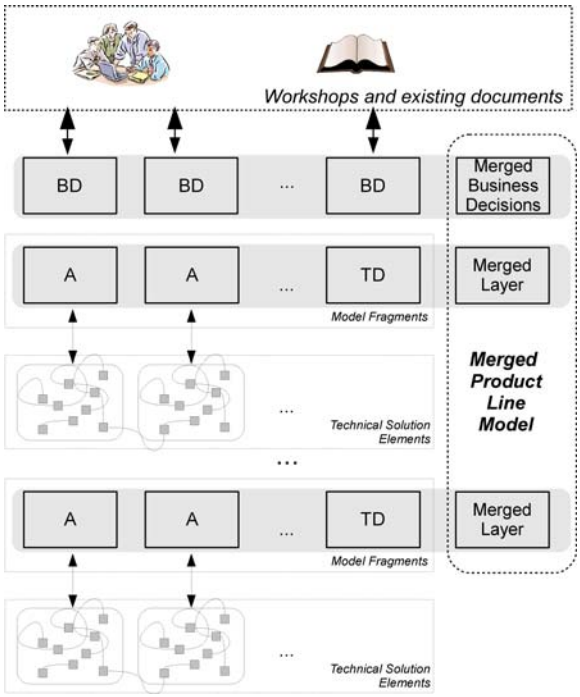


Figure 2. The improved layered approach. Fragments are defined for assets and decisions at each architectural layer. This structure also allows to separately handle business decisions (BD) and technical decisions (TD).

3.2. Refined Approach: Layered Modeling of Problem and Solution Space

We created separate model fragments for assets and decisions. Decisions models were further divided into business decision models and technical decision models. This allowed us to more clearly separate external variability (captured as business decisions) from internal variability (captured as technical decisions). In the initial approach we considered all subsystems at the same level. However, a recent refactoring of the CL2 architecture led to a clearer separation of architectural layers. We thus followed this new architectural style and created asset models and decision models in a layered fashion (see Figure 2).

We are currently refactoring the existing variability models for Siemens VAI's CL2 system following this new structure. We defined additional business decisions based on existing documents (i.e., contracts and specifications). We created initial variability model fragments for two architectural layers, one layer representing the platform for CL2 (comprising common functionality that could also be used by other systems than CL2) and one representing the CL2 system itself.

Compared to the initial approach the new structure allows to *separately manage decisions and assets* at different layers of the architecture. Changes of the underlying components can be semi-automatically reflected in the asset model fragments which helps to reduce the maintenance effort. The new approach also is more explicit regarding the *definition of stakeholder roles*. Models fragments can be assigned to designated individuals or groups to ensure their maintenance. We *used existing documents* such as technical specifications and contracts to identify business decisions. This helped to better understand the external variability of the system and to narrow the gap to non-technical stakeholders. By modeling these decisions in separate model fragments, we can achieve a clear separation of concerns based on external and internal variability.

4. Tool Support

When different people model a system from their individual perspectives they create models representing parts of the whole system. These *model fragments* are also related with each other similar to the parts of the system they represent. It is noteworthy to mention that decomposition implies recomposition [12], which means working with small models requires techniques to actually create a large model from the small ones. Model fragments are incomplete as they represent only a partial view of the system. The links to other parts of the system need to be resolved before a single model can be generated for product derivation.

4.1. Required Tool Capabilities

Product line modeling tools should provide capabilities for creating model fragments, defining inter-

fragment relationships, and integrating the model fragments. From a high-level point of view, there are two possible mechanisms for specifying model fragments and their dependencies [8].

Lazy dependencies. Modelers define placeholder elements at modeling time and assume that the references can be mapped to real elements before the models are used in product derivation. Fragments have to be explicitly merged to replace placeholders with the correct model elements. Despite the more complex merging process this approach allows to create and evolve model fragments without explicit coordination and increases the flexibility for modelers in multi-team environments.

Precise dependencies. Related model elements from other model fragments are referred to explicitly by model fragment owners when specifying dependencies between different model fragments. This requires to know the model elements of other fragments at modeling time. This can be compared to the explicit `import` statements in programming languages.

Whenever several model fragments are created at modeling time, they need to be merged before being used in product derivation. In case of placeholder references (lazy approach) dependencies between fragments are resolved manually or with the help of a tool during merging. In case of explicit references (precise approach) the merging process is easier as ambiguities have already been avoided by the modelers when creating model fragments.

4.2. Support in the DOPLER Tool Suite

As part of our ongoing tool development we have been developing an approach based on model fragments that allows defining multiple interrelated models [8]. More specifically, our DOPLER tool suite provides several capabilities to structure the modeling space. The referencing mechanism used in our tools is the lazy approach.

A model fragment in DOPLER consists of *model elements* and *placeholder elements*. Placeholder elements are introduced in a model fragment whenever relationships to elements from other model fragments need to be defined. We use concepts from programming languages to define the visibility of model elements. Modelers can specify *public elements* of fragments to make them visible outside the fragment. If model elements are internal to a subsystem with no direct relationships to elements in other models they are defined as *private elements*. The explicit location or the exact names of the referenced elements are not needed during modeling to allow loose connections between fragments. More details on this feature are described in [8].

5. Summary and Open Issues

This paper described strategies and examples of structuring the modeling space in product line engineering. There is no single optimal strategy to organize the modeling space. Our examples show that realistic strategies will be hybrid and take into account several aspects. While the strategies discussed in Section 2 provide high-level guidance it is still difficult to transfer experiences between companies and domains.

We believe that model fragments have two benefits from the perspective of variability modeling: (a) They allow to define variability locally and in a bottom-up manner (e.g., by starting at the level of teams or individual subsystems). This was the case in strategy 1. (b) They allow to define variability at different levels of abstraction and to separate those levels more clearly. This was relevant in strategy 2.

We intend to address the following research questions in our future research:

- How can we find a balance between flexibility and maintainability when structuring the modeling space? Creating many separate model fragments negatively affects maintainability while flexibility is reduced if the number of model fragments is too small. We intend to develop general guidelines depending on systems' size, structure, and complexity.
- How can we develop a mix of structuring strategies for a given context? For instance, migrating a legacy system to a product line needs a different strategy than building a product line from scratch.
- Is precise or lazy model dependency management to prefer? While developers of our industry partner preferred leeway to create and evolve model fragments this might not be the case in other organizations or domains. We plan to conduct experiments to compare the usefulness and utility of these two approaches.

References

[1] T. Asikainen, T. Männistö, and T. Soininen. A unified conceptual foundation for feature modelling. In *10th Int'l Software Product Line Conference (SPLC 2006)*, pages 31–40, Baltimore, MD, USA, 2006. IEEE CS.

[2] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-Based Product Line Engineering with UML.* Addison-Wesley, 2002.

[3] F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, and A. Vilbig. A meta-model for representing variability in product family development. In F. van der Linden, editor, *5th Int'l Workshop on Software Product-Family Engineering*, volume LNCS 3014, pages 66–80, Siena, Italy, 2003. Springer.

[4] M. Conway. How do committees invent? *Datamation*, 14(4):28–31, 1968.

[5] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, 1999.

[6] E. Dashofy, A. van der Hoek, and R. Taylor. A highly-extensible, xml-based architecture description language. In *Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, pages 103–112, Amsterdam, The Netherlands, 2001. IEEE CS.

[7] D. Dhungana, P. Grünbacher, and R. Rabiser. Domain-specific adaptations of product line variability modeling. In J. Ralyté, S. Brinkkemper, and B. Henderson-Sellers, editors, *IFIP WG 8.1 Working Conference on Situational Method Engineering: Fundamentals and Experiences*, pages 238–251, Geneva, Switzerland, 2007. Springer.

[8] D. Dhungana, T. Neumayer, P. Grünbacher, and R. Rabiser. Supporting evolution in model-based product line engineering. In *12th Int'l Software Product Line Conference*, pages 319–328, Limerick, Ireland, 2008.

[9] D. Dhungana, R. Rabiser, P. Grünbacher, and T. Neumayer. Integrated tool support for software product line engineering. In *22nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE'07)*, pages 533–534, Atlanta, Georgia, USA, 2007. ACM.

[10] T. Dolan, R. Weterings, and J. Wortmann. Stakeholders in software-system family architectures. In F. van der Linden, editor, *Second Int'l ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families (ARES'98)*, volume LNCS 1429, pages 172–187, Las Palmas de Gran Canaria, Spain, 1998. Springer Berlin Heidelberg.

[11] H. Gomaa. *Designing Software Product Lines with UML*. Addison-Wesley, 2005.

[12] R. E. Grinter. Recomposition: putting it all back together again. In *CSCW '98: Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 393–402, New York, NY, USA, 1998. ACM.

[13] G. Halmans and K. Pohl. Communicating the variability of a software-product family to customers. *Informatik - Forschung und Entwicklung*, 18(3-4):113–131, 2004.

[14] F. Heidenreich and C. Wende. Bridging the gap between features and models. In *2nd Workshop on Aspect-Oriented Product Line Engineering (AOPLE'07) co-located with the International Conference on Generative Programming and Component Engineering (GPCE'07)*. ACM Press, 2007.

[15] A. Helferich, K. Schmid, and G. Herzwurm. Product management for software product lines: an unsolved problem? *Commun. ACM*, 49(12):66–67, 2006.

[16] J. Hunt. Organizing the asset base for product derivation. In L. O'Brian, editor, *Proceedings of the 10th Int'l Software Product Line Conference (SPLC 2006)*, pages 65–74, Baltimore, MD, USA, 2006. IEEE CS.

[17] I. John. Integrating legacy documentation assets into a product line. In F. van der Linden, editor, *Lecture Notes in Computer Science: Software Product Family Engineering: 4th Int'l Workshop, PFE 2001*, volume LNCS 2290, pages 113–124. Springer, 2002.

[18] K. Jorgenson. Product modeling on multiple abstraction levels. In T. Blecker and G. Friedrich, editors, *Int'l Series in Operations Research and Management Science – Mass Customization: Challenges and Solutions*, volume 87, pages 63–84. Springer New York, 2006.

[19] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis feasibility study. Technical report, CMU/SEI TR-21, USA, 1990.

[20] K. Kang, P. Donohoe, E. Koh, J. Lee, and K. Lee. Using a marketing and product plan as a key driver for product line asset development. In G. Chastek, editor, *Lecture Notes in Computer Science: Second Software Product Line Conference - SPLC 2*, volume LNCS 2379, pages 366–382. Springer Berlin / Heidelberg, 2002.

[21] M. Matinlassi. Comparison of software product line architecture design methods: Copa, fast, form, kobra and qada. In *26th Int'l Conference on Software Engineering (ICSE'04)*, pages 127–136, Edinburgh, Scotland, 2004. IEEE CS.

[22] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. *Requirements Engineering, IEEE International Conference on*, 0:243–253, 2007.

[23] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[24] R. Rabiser, D. Dhungana, P. Grünbacher, and B. Burgstaller. Value-based elicitation of product line variability: An experience report. In P. Heymans, K. Kang, A. Metzger, and K. Pohl, editors, *2nd Int'l WS on Variability Modelling of Software-intensive Systems*, volume 22, pages 73–79, Essen, Germany, 2008. ICB Report No. 22, Univ. Duisburg Essen.

[25] M.-O. Reiser and M. Weber. Managing highly complex product families with multi-level feature trees. In *14th IEEE Int'l Requirements Engineering Conference (RE'06)*, pages 149–158, Minneapolis, MN, USA, 2006. IEEE CS.

[26] K. Schmid and I. John. A customizable approach to full-life cycle variability management. *Journal of the Science of Computer Programming, Special Issue on Variability Management*, 53(3):259–284, 2004.

[27] M. Völter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*, pages 233–242, 2007.

Functional Variant Modeling for Adaptable Functional Networks

Cem Mengi and Ibrahim Armaç
Department of Computer Science 3
Software Engineering
RWTH Aachen University, 52074 Aachen, Germany
{mengi|armac}@i3.informatik.rwth-aachen.de

Abstract

The application of functional networks in the automotive industry is still very slowly adopted into their development processes. Reasons for this are manifold. A functional network gets very quickly complex, even for small subsystems. Furthermore, concepts for variability handling are not sufficient for future standards such as AUTOSAR. In this paper we will provide an approach to reduce the size of functional networks by applying abstraction and partitioning. The achieved improvements will be described. In addition we will provide an alternative concept to handle variants in functional networks. The approach is based on extracting variation points from the functional network and modeling them separately. Open problems and future work are discussed at the end.

1 Introduction

Today the automotive industry provides customers a lot of possibilities to individualize their products. They can select from a huge set of optional fittings, e.g., parking assistant, rain sensor, intelligent light system, and/or comfort access system. The possibility to configure individual vehicles leads to the situation that both OEMs (Original Equipment Manufacturers) and suppliers have to capture explicitly potential variation points in their artifacts so that a software product line can be established to overcome the development complexity [6].

For requirements specification this is often done with so called *feature models* [4], where customer visible variants of the vehicle are captured. After requirements specification, a *functional network* is established, which consists primarily of interacting functions. Ideally, it should be used as a first concretion of the features and should help the engineer to understand the problem domain in a better way. Variation points in functional networks are captured implicitly by modeling a so called *maximal* functional network. The idea is to capture every function of the vehicle, e.g., a sensor, an actuator, a control algorithm etc., and to generate variants by removing specific parts of the functional network.

This way of variant handling is possible since the automotive software development process is hardware-dependent. A functional network is designed with detailed technical knowledge, such as the used communication infrastructure and deployment information of the functions. Therefore, deleting parts of a functional network can be regarded equally to deleting ECUs (Electronic Control Units) from the vehicle topology. We will call this kind of functional networks in this paper as *technical functional networks*.

The advantage of such an approach is that it is simple, so that development costs for variant handling can be kept down. Furthermore, virtual prototyping, i.e., behavior simulation on a PC, and rapid prototyping, i.e., real-time simulation on a controller of a rapid prototyping system or on an ECU could be almost directly adopted. This is possible because the functional network specifies the interfaces of the functions along with their communication. By implementing the behavior which is compliant to the defined interfaces a prototyping system can be set up.

Nevertheless, there are also some disadvantages. Even a technical functional network brings advantages such as simplicity and prototyping possibilities, its size gets very quickly vast so that it rather complicates the tasks of an engineer instead of supporting him. Furthermore, the automotive industry is currently performing a paradigm-shift from a hardware-driven development process to a function-driven development process, to counteract the ever increasing software complexity. The results of this efforts were specified by the AUTOSAR consortium [1]. Basically, AUTOSAR decouples the infrastructure from application software by introducing an abstraction layer between them. This implies that application software can now be developed independently from the hardware. Therefore, the methodology to capture and generate variants in a maximal functional net-

work will not be the solution for the near future.

The mentioned two problems, i.e., complexity of a technical functional network and insufficient concepts for handling variants, involve that the application of functional networks in the automotive industry is still very slowly adopted into their development processes.

In this paper we will introduce an approach to facilitate the extensive use of functional networks by reducing their complexity and providing an alternative concept for variability handling without losing the advantages such as simplicity and prototyping possibilities. The complexity reduction is primarily achieved by applying abstraction and partitioning. Abstraction is applied by identifying functions with similar semantic. This is also done for connections between functions. They are grouped to abstract functions and connections. For partitioning we identify parts of the functional network, i.e., functions and their connections which are used to model a specific characteristic of the system. For variability handling we will use a restricted form of feature models which are tailored for automotive functional networks. We will call them *functional variant models*. In this way we can extract variants from functional networks and model them separately with functional variant models. The result will be a functional network which is modeled on a logical level (in the following called *logical functional network*). Particularly, the configuration of a technical functional network from a logical functional network will be possible for utilizing the advantages of virtual and rapid prototyping.

This paper is structured as follows. In Section 2 we will describe the problems that we are dealing with in this paper. For this purposes we will introduce an example that is used for the whole paper. With the example we will describe the covered problems, i.e., the size of functional networks and variability handling. In Section 3 we will present our approaches to solve the mentioned problems. In Section 4 we will describe related work and in Section 5 we will discuss open problems and future work. Finally, in Section 6 we will summarize the paper.

2 Problem Description and Challenges

In this section we will describe the covered problems in detail. Therefore we will introduce an example, which will be used for the whole paper. Our department has gained experience on automotive software development processes by collaboration with an OEM in Germany. The example is constructed with that knowledge.

2.1 Example: Vehicle Access System

In our example we consider a *vehicle access system* for three car models. A vehicle access system is primarily a central locking system with additional features that extends a classical central locking with security and comfort features. In Table 1 we listed the features and marked those that will be supported by the appropriate model.

Table 1. Three car models and their supported features.

	Model 1	Model 2	Model 3
Active Access	x	x	x
Passive Access	x		x
Automatic Door Close			x
Selective Unlock	x		x
Anti-Theft System	x	x	
Immobilizer			x
Crash Unlock	x	x	x
Drive Lock	x	x	x

For example, an *active access* denotes a feature that supports the entry into the car by using the mechanical key or the remote control. In contrast to this, in a *passive access* a user can enter the vehicle without actively using a key. He only needs to carry the remote control with him which is extended by an identification transmitter so that the vehicle can identify the authorized user. While model 1 and 3 support both features, model 2 only supports the active access.

The *automatic door close* is a feature that closes the door automatically. The user needs only to push the door smoothly into the lock. Then the door will be completely closed by a motor that is installed inside the door.

If *selective unlock* is supported, it is possible to unlock the doors sequentially. For example, if the unlock button of the remote control is pressed once, only the driver's door will be opened. By pressing the button a second time all other doors will be opened.

The *anti-theft system* and the *immobilizer* are security features which prevent the unauthorized use of the car. While the anti-theft system blocks the steering wheel, the immobilizer additionally controls the engine and gearbox. Since the immobilizer includes in our example also the blocking of the steering wheel, these two features can only be selected alternatively.

Finally, the *crash unlock* feature unlocks the doors in crash situations, while the *drive lock* feature locks the doors after achieving a predefined speed.

If an OEM specifies the requirements for its vehicle models, commonalities and variability in the sense of the software product line paradigm are determined [6]. This has the advantage that common aspects of an artifact can be used for all models, while only variable aspects, which differentiate the products, have to be treated. Commonalities and variability for requirements specification are typically

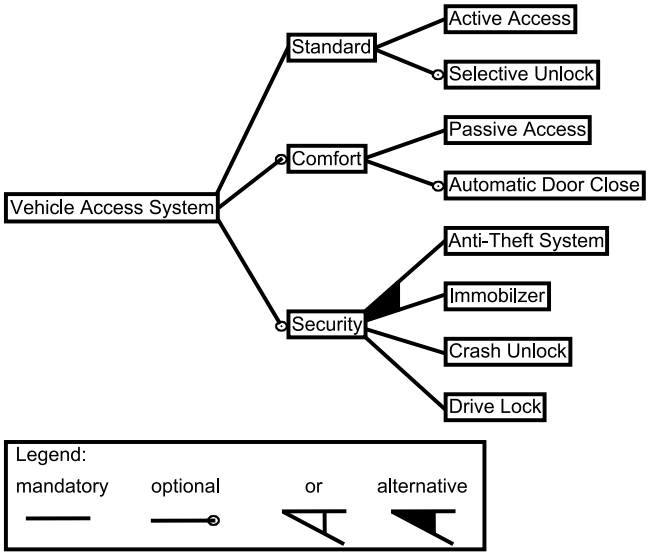


Figure 1. An example for a feature model of the vehicle access system.

captured with feature models. In Figure 1 we designed a feature model for our example. It has mandatory-, optional- and alternative-features, while or-features are not existent.

As mentioned before, OEMs currently try to cover the variability problem in functional networks by modeling a maximal technical functional network. This is an approach, which has advantages when it is applied to hardware-driven development processes. If the paradigm shift to a function-driven approach should be successfully achieved, the modeled functions must be considered on a logical level. Beside of this, it is not always straightforward to build maximal models, since there exist complex dependencies between functions. For example, there are functions with exclusive properties. In Figure 1 this is the case for the anti-theft system and the immobilizer. Furthermore, a function could require another function, so that by building a variant this constraint must be considered.

In Figure 2 we have build a maximal functional network for the vehicle access system by using the component diagram of UML (Unified Modeling Language) [2]. Functions are modeled as components and signals are modeled as connectors between ports. Incoming ports are marked white, while outgoing ports are marked black.

2.2 Size of Functional Networks

Figure 2 illustrates a functional network for the vehicle access system. We had defined 8 features, which are concretized in the functional network with about 45 functions and 113 connections. As one can see, the functional network gets very quickly very complicated, even for such a small example. We did not further modeled the anti-theft system and immobilizer in detail in order to avoid further complexity. It is obvious that a functional network for the whole vehicle is unusable.

Reasons for this complexity are multisided. First, the functional network for the vehicle access system presents a maximal model. Therefore, an engineer has to model for example both the anti-theft system and the immobilizer, even they have exclusive characteristics. Second, a system engineer models such a network on a technical level. For example, he has the knowledge about the ECU deployment of the software-based functions with the incoming and outgoing signals. Finally, the network includes redundant functions such as the five drive units with the five lock/unlock signals. For a model with two doors and a tailgate, there is no need for five drive units but rather for three. Further examples are the antennas and sensors.

2.3 Capturing Variants in Functional Networks

A further important aspect which inhibits the extensive use of functional networks for an automotive system is, that there exist only weak concepts for capturing variants. As described above, OEMs try to build a maximal functional network which is used then to derive the different variants.

This results in an enormous size of the modeled functional network (see Figure 2) and therefore inhibits the use by an engineer, because it would take more effort to understand the functional network, instead of helping the engineer in understanding the problem domain.

Considering a hardware-driven approach, building maximal functional networks are well applicable. Variants are built mainly by adding or deleting ECUs. Through the ever increasing software complexity OEMs and suppliers were forced to design an alternative approach, which allows decoupling the software from the hardware, so that a more logical view can be established in to the development process. The result of this efforts were specified in the AUTOSAR consortium [1]. Basically, AUTOSAR decouples the infrastructure from application software by introducing an abstraction layer between them. This implies that application software can be developed independently from the hardware. This in turn allows a more logical view on functional networks. Therefore we need an alternative approach to capture variation points, where the technical view on the functional network is completely ignored.

Adopting logical functional networks will bring surely a lot of advantages. Nevertheless, a technical functional network such as the one in Figure 2 has also some advantages. Particularly, the functions with their signals can be nearly directly used for virtual and rapid prototyping. To utilize

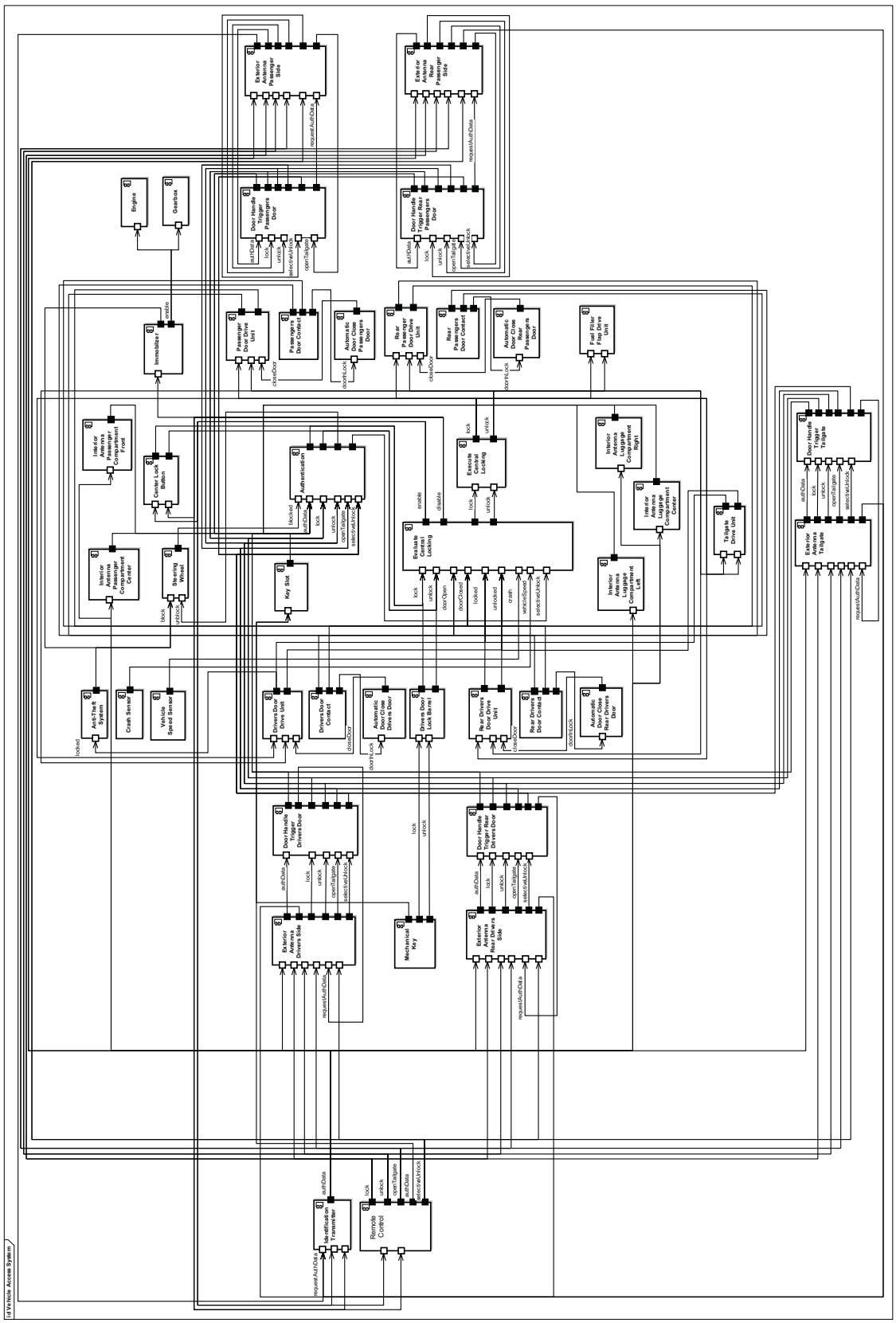


Figure 2. An example for a maximal technical functional network of the vehicle access system.

this advantage on logical functional networks we must provide an approach to configure logical functional networks to technical functional networks.

3 Functional Variant Modeling for Adaptable Functional Networks

In Section 2 we have described the problems and challenges that we are dealing with. There are mainly two core points that we have analyzed to improve the usability of functional networks in an automotive software development process. First, we have to reduce the size of functional networks to ease the work of a system engineer. He will then be able to use this artifact to understand and describe the problem domain in a better way. Second, variability handling in functional networks requires a new approach to be usable also for new standards such as AUTOSAR.

Our approach to reduce the size of a functional network is based primarily on abstraction and partitioning. With abstraction we want to achieve a more logical view on the functional network. This of course will affect the size of the functional network. With partitioning we want to divide the network into parts which belong semantically together. Examples would be an immobilizer, a central locking unit, or, if it would not be too big, even a vehicle access system. With that we want to achieve further reduction of the size.

For variability handling we will present an alternative concept for capturing variation points in functional networks. We want to extract variation points from the functional network and model them separately with so called *functional variant models*. Functional variant models represent a restricted form of feature models and are tailored for automotive functional networks. This approach provides also the possibility to configure technical functional networks for prototyping issues.

3.1 Size of Functional Networks

In Section 2.2 we have analyzed mainly three problems, which influence the size of a functional network. These are the building of a maximal model, modeling with knowledge about technical details, and finally modeling of redundant functions. To overcome these problems we propose to apply abstraction techniques for the last two aspects, and partitioning for the first one.

Considering our example in Figure 2 there are some points, where we could apply abstraction. For example, there are four functions to lock or unlock the doors, these are the *remote control*, *identification transmitter*, *mechanical key*, and *center lock button*. These four functions have all the same tasks, i.e., to control the access into the vehicle. We could replace the four functions by one function which describe exactly the task of the previous functions. For example, we could add a function called *vehicle access controller*.

Another example is given by the ten functions for the antennas which perform the task of receiving data of the *remote control* and *identification transmitter* and to transmit them to an appropriate function. The function for the *driver's door lock barrel* performs the same task for the *mechanical key*. We could replace these functions with one function to model the same task. For example, we could add a function called *data transceiver*.

The five functions for the door handle trigger in Figure 2 could be replaced by one function called *door handle trigger*.

The same technique could be applied to the drive units for the four doors, the tailgate, and the fuel filler flap. This would mean, that we could replace six functions with one function, for example called *lock/unlock drive unit*.

The functions for the door contacts could be reduced in the same manner, i.e., we replace the four functions with one function called for example *door contact*.

Finally, we do not need four functions for the automatic door close, but instead only one function called for example *automatic door close*.

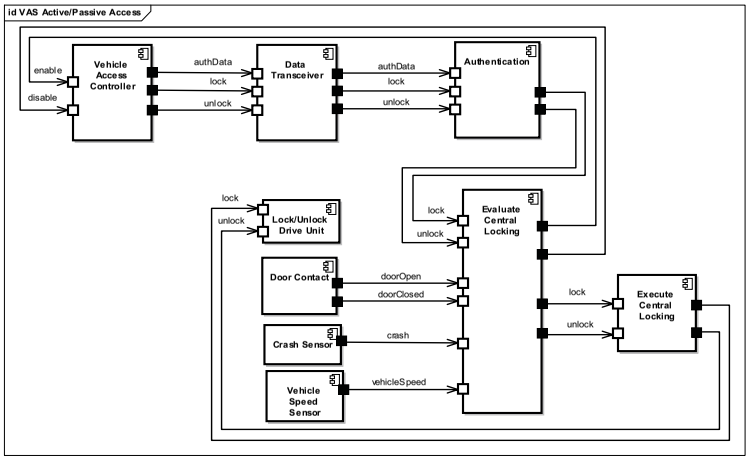
Summarizing the achievements, we see that we can replace 34 functions with 6 functions, which of course reduce the size of the functional network enormously. Note that by reducing the number of functions, we also reduce the number of redundant connections.

Another approach to reduce the size of the functional network is to divide the network into semantically equal parts, which we call *partitions*. For our example in Figure 2 this could be the *active/passive access*, *automatic door close*, *anti-theft system*, and the *immobilizer*. Note that there must not be necessarily a one-to-one mapping between the features defined in Figure 1 and the partitions.

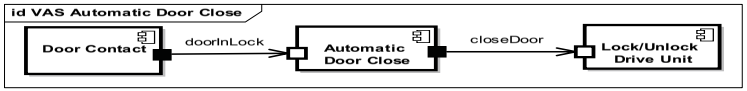
Figure 3 illustrates the result after applying abstraction and partitioning to the vehicle access system in Figure 2. Obviously, the four partitions are now more readable. We have achieved this by modeling semantically equal partitions instead of the maximal functional network for the vehicle access system. Furthermore, we use abstraction as described above, so that the size gets more reduced.

Compared to the maximal functional network, which consists of 45 functions and 113 connections, the four partitions totally has 15 functions and 27 connections. Obviously, this is an improvement.

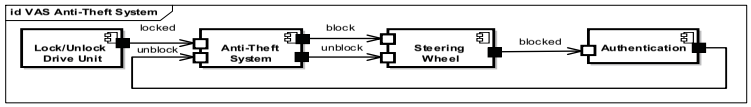
In an automotive development process which is adapted to AUTOSAR it would be ideal that such abstraction and partitioning techniques are done when the functional network is designed the first time. In that case the system engineer who has the specialized knowledge should regard such techniques. For the case that a functional network is reused the adaptation must be done by reengineering the network.



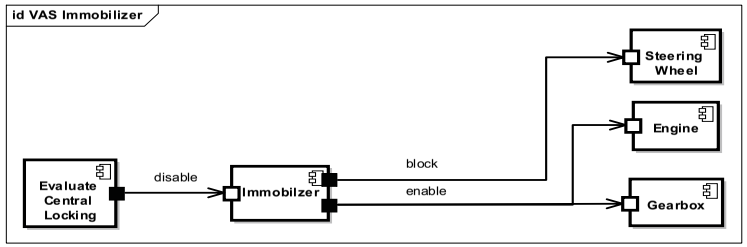
(a) Active/Passive Access



(b) Automatic Door Close



(c) Anti-Theft System



(d) Immobilizer

Figure 3. Application of abstraction and partitioning to the vehicle access system.

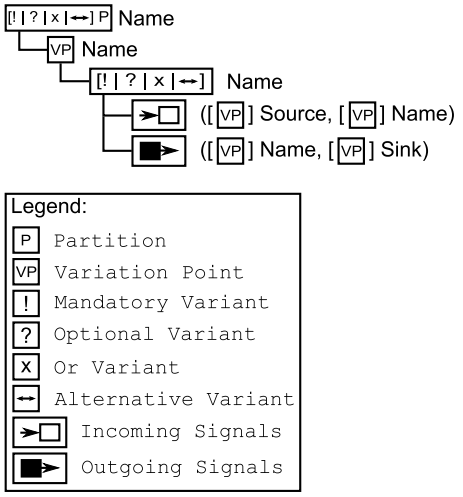


Figure 4. The structure of a functional variant model.

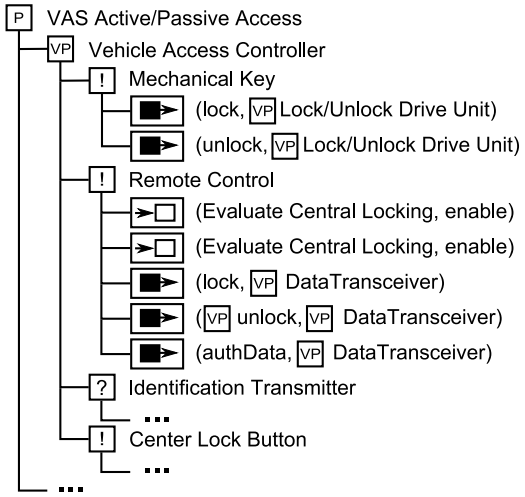


Figure 5. An example for a functional variant model for the vehicle access controller function.

3.2 Capturing Variants in Functional Networks

In Section 2.3 we have analyzed, that the automotive industry currently performs a paradigm-shift from a hardware-driven approach to a function-driven approach, to overcome the ever increasing software complexity. Particularly, the standardization of AUTOSAR is an important step towards this goal. The standard implies that vehicle functionality can be developed independently from the given infrastructure, which allows a more logical view. Therefore, the approach of modeling a maximal technical functional network to capture variation points will not be the solution for future development methodologies because it has the disadvantage that it becomes very complex and is hardly coupled to the hardware. However, it has the advantage that a technical functional network could be used in a simple way for virtual and rapid prototyping.

Adopting a logical functional network involves new requirements for modeling of variants. For example, by abstracting functions we lose information about existing variants. Therefore, we need a concept where we can gain the information back again. Furthermore, it should be possible to generate a technical functional network to utilize prototyping.

We propose an approach that is based on the concept of feature models, but is restricted and tailored for variants in logical functional networks. Variants are captured separately with *functional variant models*. In Figure 4 we have illustrate the structure. Variation points and their variants are modeled in a tree-based structure. The root consists of the modeled partition type and its name. We have defined four types, namely *mandatory, optional, or,* and *alternative*. In this way we can express the characteristic of a partition. For example, in Figure 3(b) the *automatic door close* partition is an optional partition. On the next level we have defined the variation point that is extracted from the technical functional network. A variation point can be a function or a signal. The next level contains the variant type and its name. Similar to partitions, we also have defined four variant types, namely *mandatory, optional, or,* and *alternative*. And finally, on the last level the incoming and outgoing signals are listed. Incoming signals must be denoted with its source and signal name, while outgoing signals must be denoted with its signal name and sink. The source, signal name, and sink can all be variation points. In this way, we can build a hierarchy in our models. Note that a functional variant model never exceed the described four levels and therefore enhances the visibility of variability information.

In Figure 5 we have built a functional variant model for the *vehicle access controller* from Figure 3(a). A *vehicle access controller* has mandatory functions such as the *mechanical key, remote control,* and *center lock button*. These functions are related to the *active access* feature from Figure 1. Furthermore, the *vehicle access controller* exhibits an optional function, *identification transmitter*, which allows the *passive access* into the vehicle (see Figure 1). Particularly, we have build the premises to generate a technical functional network from the logical functional network from Figure 3(a) together with the functional variant model from Figure 5. By capturing the function and signal information it would be possible, if an configuration framework is established, to regenerate a functional network that

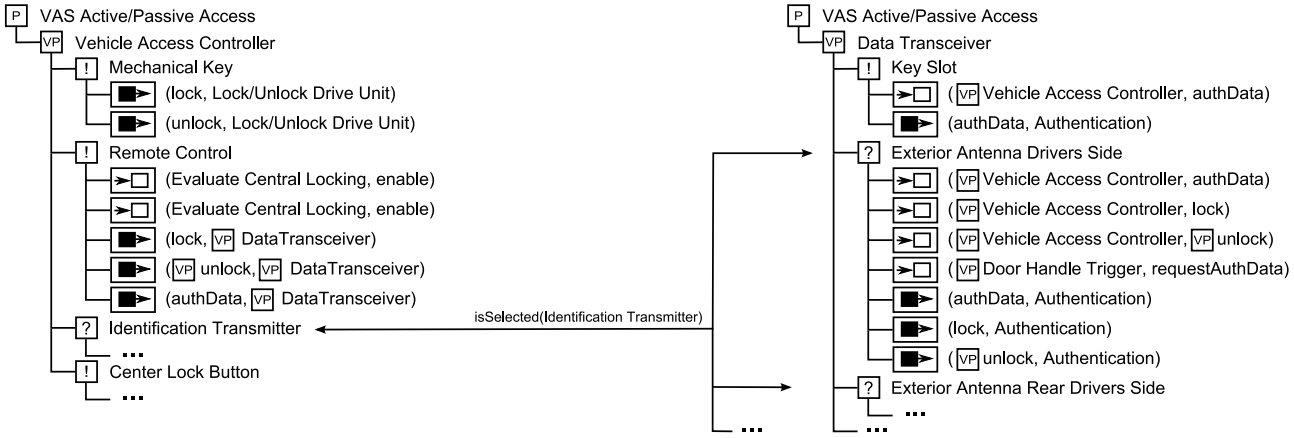


Figure 6. An example for constraints expression between functional variant models.

is ready for virtual and rapid prototyping. If we would not model the information in this way, we were not able to distinguish for example between the function *mechanical key* and the functions *remote control, identification transmitter,* and *center lock button*, which all are disabled in a crash situation (see the incoming signals of these functions in Figure 2).

Since not all configurations will be valid, we have to introduce a method which allows expressing constraints between functional variant models. An example for this is shown in Figure 6. Only if the *identification transmitter* is selected, we also have to select all exterior and interior antennas (for the *remote control* we only need the *exterior antenna tailgate*). We do not propose a specific constraint language in this paper but rather give a remark that this will also be investigated in future work.

The proposed approach allows that the variability information that was loosed when we have applied abstraction and partitioning to reduce the size of a functional network can be gained back again. Particularly, we have the premises to introduce a configuration framework that allows generating a technical functional network for prototyping purposes.

4 Related Work

There exist different techniques to model variability in a software development process. *Sinnema* and *Deelstra* give a classification about existing techniques [7]. We have adopted our approach for functional variant models mainly on feature models. In this way it will be easy to synchronize functional variant models with feature models. However, to avoid unnecessary complexity we have adopted our approach to the automotive domain.

Von der Beeck describes in his paper [8] an approach for modeling functional network with UML-RT. In our approach we did not focus on a specific architecture description language, but rather propose a way to reduce complexity and to handle variability. Particularly, we consider thereby the application to new standards such as AUTOSAR [1].

Grönniger et al. propose an approach to model functional networks with SysML to overcome the size and to capture variants [3]. Therefore, they introduce *views*. A view consists of a specific part of the functional network, such as partitions in our approach. However, in [3] a bottom-up approach is considered, i.e., it is assumed that a complete functional network exists to apply a view on it, while we propose a top-down approach to overcome the complexity in functional networks, i.e., there is no need for a maximal functional network. Furthermore, in [3] variants are also modeled with views. We propose an approach that captures variants separately with functional variant models that allow generating a technical functional network to utilize prototyping.

Kaske et al. propose a new approach to use virtual prototyping to validate functional networks [5]. While they focus on how to setup and test functional networks, we deal with the generation of technical functional networks by considering existing variants.

5 Discussion and Future Work

In this section we want to discuss open problems which have to be tackled in future work. The first question that arises is how to integrate feature models from requirements specification with functional variant models. Obviously, there is a strong relationship between these two artifacts. For example, an *active access* feature from Figure 1 is related to the *mechanical key, remote control,* and *center lock*

button functions of the *vehicle access controller*. One task thereby is to ensure the modification consistency between these two artifacts. Furthermore, a variant configuration must be properly forwarded to the appropriate artifacts and checked if it is correct. For this purposes a framework is needed that controls the consistency and the configuration between these artifacts.

In Section 3 we have seen that a functional network is divided into multiple partitions and therefore multiple functional variant models will be necessary to capture the existing variation points. An example is illustrated in Figure 6, where we have the variation points *vehicle access controller* and *data transceiver*. Note that these two variation points are included in the same partition, but they could also be modeled in the way, so that they belong to different partitions. The problem still remains the same. The decision about selecting the identification transmitter is related to the decision of selecting the optional exterior and interior antennas. Therefore there is a need to handle the consistency between all functional variant models. A constraint language would be appropriate to specify the constraints.

Since we want to generate technical functional networks from logical ones, we also have to give the possibility to relate functional variant models with functional networks in order to propagate the variant configuration information to the functional network. For example, if the functional variant model for the *vehicle access controller* in Figure 5 could be related to the logical functional network in Figure 3(a), we would have enough information to generate a technical functional network.

An obvious question that immediately arises is how this generation would be provided. Particularly, we have to ensure that incoming and outgoing signals are mapped correctly. For example, the *mechanical key* has no incoming signals compared to the *remote control, identification transmitter,* and *center lock button*. If the *vehicle access controller* is generated to a technical functional network, the signal information must be considered. Furthermore, functions such as the *door handle trigger* which have no explicit variation point in the logical functional network (compare the Figures 2 and 3(a)), must be completely inserted into the functional network.

Another important point which has to be handled is the integration of partitions. If an engineer wants to consider the functional network of two partitions that are related, we must provide a method to join them together. For example, if the *active/passive access* partition (Figure 3(a)) and the *immobilizer* partition (Figure 3(d)) should be integrated, the *evaluate central locking* functions from both partitions must be unified, and that the *disable* signals must be sent from one port.

Finally, tool support for the explained concepts must be provided. Currently, we are considering the problems on a conceptual level in order to analyze existing problems. Particularly, we believe that there is a need for tools that are tailored for the automotive domain. For example, a graphical functional network editor which only includes necessary concepts enhancing usability would help to support the application of functional networks in the development process. The same is valid for the functional variant model. Nevertheless, we must also consider the concepts on a higher level of abstract to investigate their generalization and compare them with alternatives. In this way we can make statements about the benefits of the proposed approach.

6 Summary

In this paper we have dealt with two problems, namely the complexity and insufficient concepts for variability handling in functional networks. To understand the problems in a better way we have analyzed the current development methodology. It is primarily based on modeling a maximal functional network and building variants by removing specific parts of the network. This process has the disadvantage that it is hardly coupled to the hardware infrastructure and becomes very quickly unclear.

To overcome this we have presented an approach to reduce the size of functional networks in order to support the extensive use in automotive software development. For this purpose we have applied abstraction and partitioning. The size reduction provides an improved visibility. Furthermore, the design of a functional network for a system engineer will become more simple, since technical details can be neglected. However, if a technical functional network is reused appropriate adaptations have to be done in order to build a logical functional network. For a completely new designed functional network, the adaptations should be directly regarded.

If the mentioned two techniques are applied to reduce the size of the functional network, we lose the implicit variability information. To get the information back, we have proposed the approach of functional variant models. It provides the possibility to extract variation points from the functional network and model them separately in a tree-based structure that is tailored to the automotive domain. Thus, the functional network gets more clear. Moreover, variation points are now explicitly identifiable. In addition, the functional variant model has at most four levels but allows to build hierarchies between functional variant models. It is therefore manageable and allows to integrate a structure into the models. Nevertheless, we have to consider the open problems, such as the integration with feature models, the dependencies between functional variant models, and the generation of a technical functional network from a logical one.

References

[1] AUTOSAR Website. http://www.autosar.org.

[2] Unified Modeling Language Website. www.uml.org.

[3] H. Grönniger, J. Hartmann, H. Krahn, S. Kriebel, L. Rothhardt, and B. Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *4th European Congress ERTS - Embedded Real Time Software*, 2008.

[4] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[5] A. Kaske, G. François, and M. Maier. Virtual prototyping for validation of functional architectures. In *3rd European Congress ERTS - Embedded Real Time Software*, Société des Ingénieurs de l'Automobile (SIA), Toulouse, France, 2006.

[6] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer, September 2005.

[7] M. Sinnema and S. Deelstra. Classifying variability modeling techniques. *Information and Software Technology*, 49(7):717–739, July 2007.

[8] M. von der Beeck. Function Net Modeling with UML-RT: Experiences from an Automotive Project at BMW Group. In N. J. Nunes, B. Selic, A. R. da Silva, and J. A. T. lvarez, editors, *UML Satellite Activities*, volume 3297 of *Lecture Notes in Computer Science*, pages 94–104. Springer, 2004.

Modelling Imperfect Product Line Requirements with Fuzzy Feature Diagrams

Joost Noppen
Computing Department
University of Lancaster
Infolab21, Southdrive, Lancaster
LA1 4WA, United Kingdom
j.noppen@comp.lancs.ac.uk

Pim van den Broek
Department of Computer Science
University of Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands
pimvdb@ewi.utwente.nl

Nathan Weston
Computing Department
University of Lancaster
Infolab21, Southdrive, Lancaster
LA1 4WA, United Kingdom
westonn@comp.lancs.ac.uk

Awais Rashid
Computing Department
University of Lancaster
Infolab21, Southdrive, Lancaster
LA1 4WA, United Kingdom
marash@comp.lancs.ac.uk

Abstract

In this article, we identify that partial, vague and conflicting information can severely limit the effectiveness of approaches that derive feature trees from textual requirement specifications. We examine the impact such imperfect information has on feature tree extraction and we propose the use of fuzzy feature diagrams to address the problem more effectively. This approach is then discussed as part of a general research agenda for supporting imperfect information in models that are used during variability analysis.

1 Introduction

With the increasing demand for software systems in a large variety of environments, software companies need to minimize development time and effort to remain competitive in the market. Over the years, software product lines (SPL) have become a popular means to attain these goals. In particular when families of products with overlapping functionalities must be provided, production time and cost of development can be significantly reduced by setting up an SPL infrastructure [17].

One of the most critical steps in SPL development is the identification of variable and common elements in the products that are to be supported. This stage determines which reusable assets and variability mechanisms will be included and the effectiveness of the SPL in supporting the product family. To aid software architects, many approaches have been proposed to describe and model variability and commonality of SPLs of which, arguably, feature modelling is the most well-known.

In spite of these efforts, deriving an accurate variability model from requirement documents remains a hard and complex activity. In [14] it has been identified that the vast majority of all requirement documentation is described in natural language. And as natural language is inherently inaccurate, even standardized documentation will contain ambiguity, vagueness and conflicts. Moreover, the requirements documentation of SPLs does not solely consist of standardized specifications. Rather, it consists of a range of documents that complement traditional requirement documents, such as business plans, marketing studies and user manuals.

This fragmentation of requirement documentation is already considerable for *pro-active* SPL design, when the various common assets and variations in a product line are designed upfront based on assumptions and estimates of foreseeable demands. When the variations are built into the product line incrementally driven by market needs (*reactive* SPL design) or when the product line is engineered by tailoring an existing product that is used as a baseline (*extractive* SPL design), requirements documentation becomes even more fragmented and inaccurate. Requirements can originate from various unrelated sources and can initially be specified without an SPL context in mind. The specifications that result generally are unfit for accurate SPL development as the ambiguous and unclear definitions hinder

the accurate representation and analysis of variability and commonality.

In this article, we examine the influence such *imperfect information* has on the definition of variability models and its consequences during SPL development. In particular, we examine the stage that tries to derive feature trees from requirement documentation. Based on the identified problems, we define an approach and research agenda for dealing with imperfect information during feature tree definition more effectively. To illustrate these results, an approach [2] that derives feature trees by clustering requirements based on their similarity is taken as a case study.

The remainder of this article is as follows. In Section 2, the problem of imperfect information during SPL development is defined. Section 3 explores the impact of imperfect information on approaches that cluster requirements to form feature trees and analyses how imperfect information influences their effectiveness. Our approach for capturing imperfection during feature tree derivation is described in Section 4 and in Section 5 we discuss and define the research agenda for imperfect information support in variability modelling. Related work is discussed in Section 6 and Section 7 concludes.

2 Imperfect Information in SPL Development

2.1 Introduction

Successful SPL development requires a considerable amount of information on, for example, the products to be supported, expectations of the product line architecture, etc.. But as we have identified in Section 1, the requirements documentation generally contains vagueness, ambiguities, conflicts or information is missing completely. For this reason, the available information might not be fit as input for the design of an SPL.

To demonstrate the impact imperfect information can have on SPL development, consider a software product line of home automation systems. These systems are designed to automatically co-ordinate various devices present in the users home in order to regulate lighting, heating, fire control and various other concerns via centralised control by the inhabitant. The following excerpt originates from the requirement definition that originally appeared in [1].

1. Access control to some rooms may be required. The inhabitants can be authenticated by means of a PIN introduced in a keypad, passing an identification card by a card reader, or touching a fingerprint reader.
2. If the opening of a window would suppose a high energy consumption, because, for instance, the heater is trying to increase the room temperature and it is too cold outside, the inhabitants will be notified through the GUI.

When examining these descriptions, it can be seen that the information provided can hinder successful SPL development. There are a number of statements that contain ambiguity. For example, requirement 1 states that "some" rooms "may" require access control. They word *may* in this context can mean that it is optional to have access control for a *product* of the SPL, but it can also mean that it is optional to be included in SPL all together. The second requirement states that a check should be made on whether opening a window leads to "high" energy consumption. It does not specify what *high energy consumption* exactly means and whether this is defined by an inhabitant or it should be determined by the Smart Home.

When this information is used during SPL development, the software architects will use the interpretations that seem most logical. However, the interpretation that is chosen will make a huge difference for the resulting SPL. When high energy consumptions is to be determined by the inhabitant, it will influence user interaction elements. However, when this is to be determined by the Smart Home, it will require changes to sensors and reasoning elements.

The traditional way of solving imperfection is to resolve it together with the stakeholders [5]. However, such kind of perfective modifications require assumptions on what is actually meant by the imperfect information. When these assumptions can be justified, this is a valid step during the design. When such assumptions cannot be justified, however, the imperfection can only be resolved by making an arbitrary choice between the possible interpretations.

Nonetheless, the choice of one interpretation over the other can have considerable impact on the SPL and its effectiveness. In particular the phase in which requirements are clustered into features (whether manually or by a tool), the consequences can be severe. The choice of interpretations directly impacts the structure of the resulting feature tree which is used to determine the common assets and variability mechanisms of the SPL, two elements that are key for the success of the SPL.

2.2 Definition of Terms

To establish a uniform terminology, we define the concept of imperfect information in terms of information being *sufficient* or *insufficient* with respect to the context in which it is used:

- Information is considered to be *sufficient* with respect to a particular context of use when it is possible to

come to an optimal decision in this context with only this information.

- Information is considered to be *insufficient* with respect to a particular context of use when the information is not sufficient for this context.

The concept of sufficiency relates information to the context in which it is to be used. Depending on decisions to be taken, the available information can or can not suffice to resolve the situation. For example, when the uncertain occurrence of an undesired event is given by a probabilistic model, this information is sufficient to perform a risk analysis. However, the probabilistic model does not provide sufficient information to answer the question *"Will the event occur tomorrow?"* with a yes or no. Whenever information is insufficient for particular context of use, this can be resolved in two ways: perfecting the information to make it sufficient for its context or adjusting the context so the information becomes sufficient.

Based on these definitions, perfect and imperfect information can be defined as follows:

- Information is considered to be *perfect* when it is sufficient in any context of use.
- Information is considered to be *imperfect* when it is not perfect.

3 Feature Trees from Imperfect Textual Requirements

3.1 Requirements Clustering

Since its first introduction [10], feature-oriented domain analysis has become one of the most widely used variability-modelling approaches. Over the years, many extensions have been proposed to this initial model, such as FeatuRSED [8], and it has been successfully applied in the design of software product lines. With increased use, the formal semantics and correctness of feature diagrams has also received increasing attention [3, 19]. But while the expressiveness and correctness support for feature diagrams has significantly increased, systematic approaches for deriving feature trees from requirement specifications have been few and far between.

Rather than a systematic process, the derivation of an initial feature tree from the provided requirement descriptions has remained something of a black art. The successful definition of a feature tree that accurately represents the information in the requirement specifications still depends heavily on the intuition and experience of the software architect. Assistance for this process has been proposed, but this mostly consists of guidelines and heuristics [13, 6]. Nonetheless, these approaches acknowledge the difficulty of this step as the vagueness in both requirement descriptions and the understanding of what exactly constitutes a feature severely hinders the systematic derivation of feature diagrams.

Generally, the initial feature tree structure is determined by performing a clustering on the requirements. Requirements that relate to the same concepts are grouped together in clusters. The clustering that results forms the basis for the features in the feature tree. However, clustering-based approaches (implicitly) expect requirement specifications to be clear, structured and unambiguous. Requirements can only be clustered accurately and effectively if they are accurate and free of conflicting and ambiguous information. In practice, however, the provided requirement specifications seldom exhibit these properties.

Ambiguous textual specifications can lead to different clusterings depending on how the requirement is interpreted. It can be said therefore, that such requirement specifications provide *insufficient* information for accurate clustering and feature tree derivation. Most times, this problem is "resolved" by making explicit assumptions on the meaning of the imperfection information, even when this can not be justified. As a result, the resulting feature trees are based on unjustifiable information and can have a significantly different structure than would have been the case otherwise.

Nonetheless, the resulting feature trees are used for decisions that are critical for the success of the SPL, such as the choice of core assets and variation mechanisms. And as an SPL has a significantly longer lifecycle than traditional software systems, the consequences of wrong decisions will also be felt longer. It is therefore vital that imperfect information in requirement specifications is identified and its influence is well understood. Rather than replacing imperfection with unjustifiable assumptions, the nature and severity of the imperfection should be modelled and considered during the definition of feature trees.

In the following section, we examine the impact of imperfect information for a clustering approach that derives feature trees from textual requirements documentation by clustering requirements based on their similarity. We identify where imperfect information can manifest itself and how it influences the effectiveness of the approach. In Section 4, we propose an approach for handling imperfection in textual requirements and representing it accordingly in feature trees.

3.2 Similarity-based Requirements Clustering

As indicated in the previous section, clustering of requirements is used as a basis for the identification of features that make up a feature tree. Naturally, the relevancy of

the resulting feature tree heavily depends on how requirements are clustered into features. In [2], it was argued that such clustering can be based on the measure of similarity between requirements, as similar requirements typically relate to the same concepts and therefore likely belong to the same feature. The continued work of [2], called Arborcraft, extends on this notion by defining an approach that clusters textual requirement specifications based on similarity of requirements.

3.2.1 Overview of the Approach

The Arborcraft approach clusters requirements together based on the similarity these requirements exhibit from their natural language descriptions. From the clustering that results, a feature tree is derived. In Figure 1, the phases of Arborcraft are depicted.

In stage I, the similarity of requirements expressed in natural language is determined using *latent semantic analysis* (LSA) [20]. Without going into detail, LSA considers texts to be similar if they share a significant amount of concepts. These concepts are determined according to the terms they include with respect to the terms in the total document space. As a result, for each pair of requirements a *similarity measure* is given, represented by a number between 0 (completely dissimilar) and 1 (identical). In the figure, this result is represented by the block *Requirements Similarity Results*.

Stage II uses the *Requirements Similarity Results* and a hierarchical clustering algorithm to cluster requirements that are semantically similar to form features. Small features are clustered with other features and requirements to form parent features. The similarity measure is therefore used to build up the hierarchy of the feature diagram.

Finally, in stage III Arborcraft provides the possibility to identify variability and crosscutting concerns in the requirements documents. Based on these results, the feature tree can be refactored by, for example, relocating requirements or introducing aspectual features.

Consider the similarity analysis result in Table 1. In this table, the similarity of four requirements has been determined, R_1, R_2, R_3 and R_4. After the clustering stage of Arborcraft, the feature tree of Figure 2 results. Requirements R_1, R_2 and R_3 are clustered together on the second level of the feature tree, as they are the most similar. As a result of the lower similarity, R_4 is clustered with the other requirements only on the highest level. Due to a lack of space, the refactorings of the final stage have been omitted.

3.2.2 Imperfect Information in Arborcraft

Arborcraft provides a comprehensive approach for extracting feature trees from textual requirements specifications. However, early experimentation has indicated that small, well-structured and clear documents produce considerably

Table 1. Requirement Similarity Values

	R_1	R_2	R_3	R_4
R_1	1	0.9	0.6	0.4
R_2	0.9	1	0.8	0.6
R_3	0.6	0.8	1	0.4
R_4	0.4	0.6	0.4	1

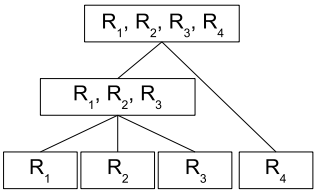


Figure 2. Feature Tree

better results than large, unstructured documents containing vagueness and ambiguity. This supports our claim in Section 3.1 that such insufficiency can severely hinder the effectiveness of clustering-based derivation of feature diagrams. The impact of imperfect information on the Arborcraft approach can be identified in the following areas:

Requirements clustering
Imperfect information influences the step of clustering requirements. The similarity of requirements is determined by evaluating the amount of concepts that are shared between them. But as indicated before, natural language naturally contains ambiguities and depending on which interpretation is compared to the other requirements, the results of the similarity analysis will differ. It is therefore vital that the similarity analysis is provided with clearly defined specifications if it is to provide accurate results.

Feature tree derivation from clusters
For the step from clusters to feature trees, there is no direct influence of imperfect information. However, this step uses the clustering result of the previous step and assumes that these results are accurate and reliable. Moreover, this stage aims to come up with a single feature tree, which is realistically speaking neither possible nor desirable given the presence of imperfect information. By having to commit to one particular feature tree, the software architect is forced to commit to the chosen interpretations for the identified imperfection in the clustering step.

Variability and crosscutting analysis
The final stage of Arborcraft searches for variability and crosscutting by analysing the requirements based on their clustering in the feature tree. This stage uses semantic anal-

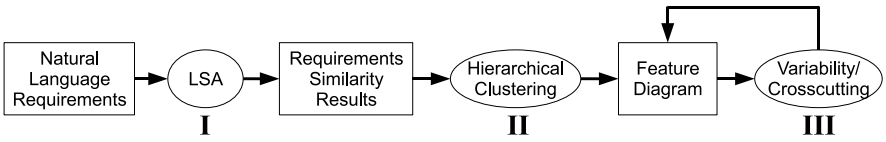


Figure 1. The stages of Arborcraft

ysis of the textual requirement descriptions, which means the imperfection in natural language naturally influences the effectiveness of the result. Again, unjustifiable assumptions can be required to come up with refactorings for the feature tree. Moreover, this stage does not ensure that for both the clustering and the proposed refactorings the same interpretations and assumptions have been considered.

4 Imperfection Support for Feature Tree Derivation

4.1 Introduction

We propose a coherent approach for handling imperfect information during the derivation of feature trees from textual requirements. The main element of the approach is the *fuzzy feature diagram*, an extension to traditional feature diagrams that can capture ambiguous requirements and describe how they can be part of multiple features simultaneously. This extension is then used to accommodate the influence of multiple interpretations for ambiguous information in the textual requirement documents. Our approach consists of three steps:

1. Model imperfection in textual requirements
2. Clustering of requirements including imperfection
3. Derivation of a *fuzzy feature diagram*

Our proposed approach focuses on *ambiguous* information, i.e. information that can be interpreted in multiple ways. The first two steps are aimed at handling the multiple interpretations of ambiguous informationthat is found in textual requirement documents. The third step uses *fuzzy feature trees* to describe the clustered ambiguous information, rather than attempting to describe this information using traditional feature diagrams. In the following sections, we describe these steps in more detail.

4.2 Modelling Ambiguous Requirements

The first extension is explicit modelling of ambiguous information in the textual requirement specifications. For all identified ambiguities, instead of considering a single interpretation, all relevant interpretations are described. Moreover, each interpretation is attributed with a *relevance value*, a number between zero and one that indicates the perceived relevance of the interpretation. The single imperfect requirement therefore is replaced by a *fuzzy set*[1] of interpretations.

Requirements that are defined using fuzzy sets of interpretations are called *fuzzy requirements* and were first proposed in [15]. The identification of the interpretations and the definition of their relevance values will be done in close cooperation with the stakeholders. In subsequent steps, all identified interpretations are included in the development process as normal requirements. Note that, naturally, ambiguities need to be identified before they can be modelled, but this goes beyond the scope of this work.

To illustrate this step, in Section 2.1 we indicated that the *high energy consumption* defined in the Smart Home requirements can mean "pre-defined by the habitant", but can also mean "a measurement result from the system". With the extension proposed above, this imperfect requirement is replaced with $\{p$/*"exceeds a pre-set energy consumption level", q*/*"when the system determines a high energy consumption level"*$\}$, where p and q are the respective relevancy values of the interpretations. The ambiguous statement is now refined to two explicit interpretations, which both can be considered in subsequent steps.

4.3 Clustering of Ambiguous Requirements

In Section 3.1, we have established that due to the use of natural language generally provides insufficient information to determine a single best clustering of requirements. With the concept of fuzzy requirements, this problem can now be resolved. By considering all interpretations as tra-

[1] Fuzzy sets allows elements to be a partial member of a set, which can be used to describe imperfect information. The partial membership of an element x in a fuzzy set is given by the membership value $\mu(x)$, where μ is a function that maps the universe of discourse to the interval $[0, 1]$. This value is the degree to which x is an element of the fuzzy set, where 1 means "completely a member" and 0 means "completely not a member". By considering membership degree during manipulations, the risk and impact of the modelled imperfect information can be assessed more accurately. Due to the lack of space a more elaborate introduction is omitted, but the interested reader is forwarded to [12].

ditional requirement, the clustering of requirements can be performed in much the same way as before. However as a result, different interpretations can end up in different clusters, even when they originated from the same fuzzy requirement. This essentially means that the fuzzy requirement has become a member of multiple clusters simultaneously, which is not possible with traditional clustering of requirements.

We therefore propose to use to group requirements into fuzzy clusters (fuzzy sets) instead of traditional, crisp clusters. Requirements can be part of multiple clusters at the same time and to differing degrees. We define the degree of membership of an interpretation in a fuzzy cluster to be the relevancy degree of that interpretation in the fuzzy requirement. A fuzzy clustering is then achieved by first clustering all crisp requirements and interpretations using a traditional clustering method. Then, in all the clusters that contain interpretations, these interpretation are replaced by the original imperfect requirement and they are tagged with the relevancy degree of the interpretation that was replaced, thus creating fuzzy sets.

Consider requirements R_1, R_2 and R_3, where the imperfect requirement R_3 is replaced with the fuzzy requirement $x/R_{3.1}$, $y/R_{3.2}$. A traditional clustering of crisp requirements and interpretations has resulted in the clusters R_1, $R_{3.1}$ and R_2, $R_{3.2}$. Here, the requirement R_3 has become part of two clusters due to two different interpretations. This clustering is transformed into a fuzzy clustering by replacing the interpretations in the clusters with the initial imperfect requirement: R_1, x/R_3 and R_2, y/R_3. As indicated, the membership values correspond to the relevancy degrees of the respective interpretation.

4.4 Fuzzy Feature Trees

Where traditional, crisp clustering leads to the definition of a feature diagram, with the proposed fuzzy clustering this is no longer possible. Therefore, our third proposed extension is the use of *fuzzy feature trees*. A traditional feature tree forces the software architect to precisely nail down variability and hierarchical structure for the software product line. A fuzzy feature tree does not expect this kind of precision.

In a fuzzy feature tree, requirements are clustered into features to a certain *degree*, which means that features in a fuzzy tree are *fuzzy clusters*. Moreover, a fuzzy feature tree imposes no restrictions on features or relationships between them, even when this would be invalid in traditional feature trees. For instance, multiple features can contain the same requirements and it is possible for a feature to have multiple parents.

In Figure 3, a fuzzy feature tree is depicted that is a modification of the diagram in Figure 2. In this picture, the requirement R_3 initially was identified as being ambiguous and two interpretations were been identified, say $R_{3.1}$ and $R_{3.2}$, with a respective relevancy degree of x and y. In the fuzzy clustering that results, there are two overlapping requirement clusters, $\{R_1, x/R_3\}$ and $\{R_2, y/R_3\}$, in which R_3 has differing degrees of membership. This fuzzy feature tree now describes the best clustering that could be achieved in the feature tree based on the ambiguous information that was provided. Note that as a result of including these clusters, other features have multiple parent features.

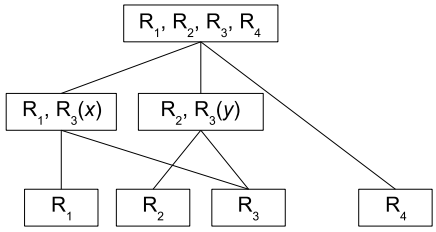


Figure 3. A Fuzzy Feature Diagram

As a fuzzy feature diagram is a generalization of feature diagrams, it essentially describes multiple feature diagrams simultaneously. By removing elements, such as features with overlapping requirements, a fuzzy feature diagram can be transformed into a traditional feature diagram. Depending on how the membership degrees are used during this *defuzzification* step, a number of alternative feature diagrams can be proposed to the software architect. Ideally, however, a fuzzy feature diagram is maintained throughout SPL development so more detailed information can arrive at a later stage to resolve the imperfection. Also, by extending approaches that operate on traditional feature diagrams, such as the variability/crosscutting analysis of Arborcraft, they can assess the risks that come with specific decisions by considering the imperfection described in fuzzy feature diagrams.

4.5 Application to Arborcraft

When this proposal is applied to the Arborcraft approach, this results in the picture of Figure 4. The new elements in the picture when compared to Figure 1 are indicated in grey. In step (I), first the ambiguous statements are modelled as fuzzy requirements. The resulting requirements are then clustered with the standard techniques from Arborcraft (steps II and III). The feature tree that results is restructured to a fuzzy feature tree in step IV. If required, in step VI the fuzzy feature tree is defuzzified to a number of crisp feature trees. The software architect can then select the most appropriate alternative. The refactorings of Arborcraft with respect to variability and cross-cutting (step V) can be applied to both fuzzy feature tree as well as the

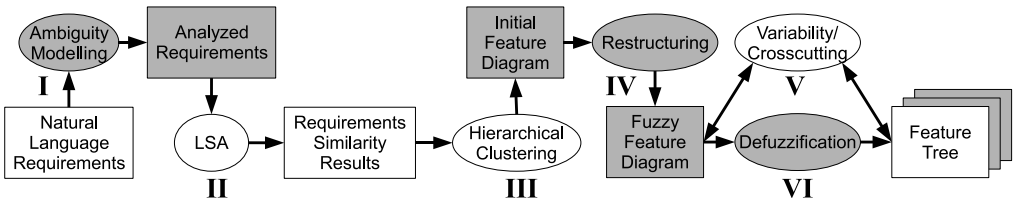


Figure 4. Arborcraft with support for ambiguous requirements

defuzzified, crisp feature trees. Naturally, to handle fuzzy feature trees the refactoring mechanisms would need to be enhanced.

5 Discussion

5.1 Discussion of the Approach

In the previous section, we have sketched an approach for coping with imperfect information during the definition of feature diagrams. Nonetheless, this approach leaves a number of questions unanswered. In this section we examine some of these questions.

5.1.1 Maintaining Imperfect Information during SPL Development

One of the key properties of the proposed approach is the possibility to maintain imperfection during multiple stages of SPL development. With this property, software architects do not need to make unjustifiable assumptions which can hinder development at later stages. However, the inclusion of alternative interpretations creates extra overhead and introduces new model elements that need to be considered.

However, the benefit of maintaining imperfect information during SPL development is two-fold. First, the presence of imperfect information poses a danger to effective software development. If the SPL architecture is built based on imperfect information, it is likely that at later stages some sort of redesign will be required. By including alternative interpretations and considering the influence of imperfection, the design becomes more resilient to such changes. In essence, design becomes a defensive activity as a number of likely scenarios are already considered. Moreover, the modelled imperfection offers the opportunity to examine risks that come with design decisions.

The second benefit lies in the fact that insufficient information might not stay insufficient indefinitely. A traditional approach forces the architect to make explicit assumptions that at a later stage can turn out to be correct or false. With the support for imperfect information, the architect does not have to commit to a single assumption. Rather, the design will consider and support a number of alternative assumptions throughout the design stages. This creates a larger window of opportunity for the imperfection to be resolved and it will require considerably less refactoring if the design already contains what turns out to be the correct information.

Nonetheless, the concept of maintaining imperfect information introduces a trade-off of effort during the development process. By including all kinds of potential interpretations and design alternatives, software architects can be faced with a considerable increase in effort. It is therefore vital that the software architect can control and manage the extra effort that is created. This can be achieved by, for instance, focusing only on the most relevant interpretations and removing excess information whenever possible. To perform this kind of operations, support for (partial) removal of imperfect information from design is required.

5.1.2 Identifying Imperfect Information

One of the most important problems to be solved is the identification of imperfect information in textual requirement specifications. More specifically, the sufficiency of available information needs to be determined for the design step in which it will be used. As the reason for information to be insufficient is defined by this context, imperfection not only needs to be identified, but also the nature and consequences of its insufficiency.

NLP techniques can assist in identifying imperfect information in textual requirement specifications. Specific approaches have been proposed for the identification and management of imperfection in requirement specifications, such as [9, 11]. With the addition of specific lexicons and vocabulary for typical imperfection in software specifications, semantic analysis approaches can be extended to aid in this goal.

Ideally, at the moment imperfection is identified the stakeholders are consulted and additional information is acquired to resolve the situation. However, as we identified in Section 2, this is not always possible due to a lack of knowledge or insight. In this case, stakeholders can be consulted to describe the actual imperfection. For example, in our extension for requirements clustering we propose alternative

interpretations to be given for ambiguous statements. While NLP cannot derive this kind of information, the automatic identification of potential ambiguities provides a valuable first step.

5.1.3 Removing Imperfection Models from SPL Development

As mentioned in the previous section, it can be desired at given points during development to remove imperfection models. This can for instance be the case when subsequent stages no longer support imperfect information or when the additional effort for maintaining it are no longer feasible. This warrants the question how these models can be removed from the design when the need arises.

As the proposed extensions, such as the fuzzy feature diagram, essentially describe a number of traditional models using a single generalized model, the removal of imperfection corresponds to determining the best traditional model from the generalized model. The numerical information (such as the membership values of clustered requirements) can be used for this purpose. By identifying the traditional, crisp model that best fits the numerical information and the restrictions the model should adhere to, the imperfection can be removed. This is in essence an optimization problem where all traditional models that can be derived from a generalized model are ranked and the best one selected.

5.1.4 How do Imperfection Models affect existing Approaches?

The introduction of fuzzy feature trees in this paper, and imperfection models in general, has a direct impact on all approaches to operate directly on traditional feature trees. As these approaches do not consider the typical properties of fuzzy feature trees, they cannot be applied in the same manner during SPL development.

This can be resolved in two possible ways: first of all, the imperfection can be removed at the moment a design activity is to be undertaken that does not support imperfection models. This can be done by the earlier mentioned defuzzification techniques and the selection of one of the resulting alternative feature trees. However, as identified in Section 5.1.1, it is desirable to maintain unresolved ambiguity as long as possible. Therefore, the second way is to extend these approaches to support fuzzy feature trees. Naturally, the effort required for realising such support must be aligned with the benefits during development.

5.2 A Research Agenda for Supporting Imperfect Information

In Section 2, we have identified that imperfect information in textual requirement specifications can severely hinder variability/commonality analysis. And with the proposal of fuzzy similarity values, overlapping clustering and fuzzy feature trees, we have sketched a first direction for imperfection support in Arborcraft. In this section, we define a research agenda with key problems of imperfect information in feature tree derivation and the general direction on how these problems should be addressed.

A formalized procedure for deriving feature diagrams
One of the most important problems is a complete understanding of the process that turns textual requirement specifications into actual feature diagrams. At the moment, this step still relies considerably on the intuition, knowledge and experience of the software architects. To understand how imperfect information influences the decisions that define the feature tree, they need to be understood in an unambiguous and uniform manner.

With a formal model of the derivation of feature trees, the way information is used will become well-understood. Moreover, it becomes possible to analyse the problems that imperfect information causes during this process. NLP-baed approaches such as Arborcraft actually define a (semi-)formal approach for going from textual requirement specifications to feature diagrams. The proposed approach in this article utilizes this by extending its capabilities to support imperfection.

A taxonomy of types of imperfect information
A second important problem to be solved is understanding the nature of the imperfection that can occur in requirement specifications. Many different types of imperfection exist, such as conflict, ambiguity and vagueness, and each of them influences the usability of the information in a different manner. By having a standardized categorization of imperfection types, potentially hazardous elements in specifications can be identified and its impact assessed.

In particular for NLP-based approaches, the definition of an imperfection taxonomy would be very useful. As many NLP approaches utilize a semantic lexicon (e.g. EA-Miner uses variability lexicons), an imperfection lexicon based on this taxonomy can aid in the automatic identification of imperfect information. Moreover, a well-defined terminology will aid in communicating about imperfect information and creating awareness about this phenomenon.

Modelling and reasoning support for imperfection
The core element of any approach for dealing with imperfect information is the ability to model the imperfection and reason with this model in subsequent steps. By capturing the identified imperfection with models such as probability theory and fuzzy set theory, the nature and risk of such information can be quantified. By extending the subsequent design steps to consider these models, the influence of the

imperfection can be considered during decision making activities.

Resolving this problem requires the first two problems of this research agenda to be resolved. Formalizations need to be extended with techniques from probability and fuzzy set theory to support reasoning with models for imperfection. Moreover, only when the type and nature of the imperfection is known is it possible to identify the appropriate model to quantify it accurately.

Removal of imperfection models from development
The final problem is the systematic removal of imperfection models from development. The first three research problems are targeted at introducing models for imperfection. Conversely, at particular stages and situations it can be required to remove these models because of new insights or because of a lack of budget to maintain all the extra information.

The removal requires an approach that can determine which elements of the imperfection models are no longer relevant. Moreover, it can also require the selection of the most relevant interpretations that have been included. As identified in Section 5.1.3, such removal activities are in essence optimization problems so the body of knowledge in optimization theory offers a promising starting point.

6 Related Work

This paper focuses on the problem of supporting imperfect information in feature derivation and relates to the areas of requirements engineering, SPL development and imperfection modelling and support for feature diagrams. In this section, we give a short overview of related work in these fields.

Extensions to feature diagrams based for imperfect information have been proposed before. In [18], features in feature diagrams are attributed with fuzzy weights that describe the preference of specific customers. The weights subsequently are used by a fuzzy logic-based expert system that can determine typical product configurations for specific customer profiles. The approach described in [16] extends on this approach by introducing fuzzy probabilistic descriptions of market conditions that can influence the expert system. In [7], soft constraints are introduced that specify the conditional probability that feature will be part of a configuration when another feature already is part of it. This information can then be used to identify product parts that must be supported by a particular platform or to understand how these products utilize a particular platform.

These approaches capture a particular type of imperfection when dealing with feature diagrams. However, the goal of these approaches is distinctly different from the problem we have identified in this article. The imperfection that these approaches support originates from an uncertain preference of customers for particular configurations. The imperfection support in our approach addresses unresolvable imperfection in requirement specifications. Moreover, our approach is aimed at supporting the design steps that lead up to an actual feature diagram.

In this work, a fuzzy extension is proposed for deriving feature diagrams from textual requirement specifications. In [4], an approach is proposed that derives feature trees by performing clustering of textual requirements definitions. Our approach is a generic extension that can be integrated in this and other similar approaches, like Arborcraft. Imperfection support for feature tree extraction, to the best of our knowledge, has not been proposed before.

The influence of imperfect information on feature diagrams is well recognized [18, 16]. Kamsties identifies in [9] that ambiguity in requirement specifications needs to be understood before any subsequent design can be undertaken. With support for feature tree definition being largely heuristic [13, 6], systematic support for imperfect information is all but completely absent. Our approach defines a first step by proposing models and design steps to support these models.

7 Conclusions

In this article, we have taken a first step towards supporting imperfect information in the definition of feature trees. We have identified that the effectiveness of approaches that derive feature trees from textual requirement specifications can be severely compromised by imperfect information. And as imperfection is naturally part of requirements that are specified in natural language, its influence on such approaches can not be ignored. We established that the main cause is that most approaches require perfect information for the definition of an accurate feature trees. As a result, any imperfections need to be resolved even when specific assumptions can not (yet) be justified.

To illustrate the impact of imperfect information, we explored approaches that derive feature trees from requirements specifications by clustering related requirements. As the clustering mechanisms used in these approaches do not explicitly consider imperfection, the clustering that results is influenced by vagueness and ambiguity in natural language. Nonetheless, subsequent stages use the feature tree that results as input while assuming these results to be accurate.

To address these problems, we have proposed an approach that captures ambiguity in requirement descriptions using techniques from fuzzy set theory. In particular, we proposed the consideration of multiple interpretations when ambiguity can not be resolved, fuzzy clusters to extend the clustering of requirements and a fuzzy extension for feature

diagrams that captures the imperfection. These proposals have been generalized to form a research agenda for imperfection support in feature diagram derivation.

As future work, we want to formalise the steps for the derivation of fuzzy feature trees from ambiguous requirements. Also, we want to integrate with existing approaches that can identify imperfect information using natural language processing and we want to extend the support for refactorings of feature diagrams. When this is completed, we plan to implement the approach as part of Arborcraft and evaluate it with an industrial case study.

8 Acknowledgements

This work is sponsored by the European Union as part of the AMPLE project (IST-33710) and the DISCS project (IEF-221280).

References

[1] M. Alvarez, U. Kulesza, N. Weston, J. Araujo, V. Amaral, A. Moreira, A. Rashid, and M. Jaeger. A metamodel for aspectual requirements modelling and composition. Technical report, AMPLE project deliverable D1.3, 2008.

[2] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler. An exploratory study of information retrieval techniques in domain analysis. In *SPLC '08: Proceedings of the 2008 12th International Software Product Line Conference*, pages 67–76, Washington, DC, USA, 2008. IEEE Computer Society.

[3] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering*, volume 3520 of *Lecture Notes in Computer Science*, pages 491–503. Springer-Berlin, 2005.

[4] K. Chen, W. Zhang, H. Zhao, and H. Mei. An approach to constructing feature models based on requirements clustering. *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, pages 31–40, 2005.

[5] A. Classen, P. Heymans, and P.-Y. Schobbens. What's in a feature : A requirements engineering perspective. In *LNCS 4961*, pages 16–30. Springer-Verlag, 2008.

[6] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[7] K. Czarnecki, S. She, and A. Wasowski. Sample spaces and feature models: There and back again. *Software Product Line Conference, International*, 0:22–31, 2008.

[8] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the rseb. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 76, Washington, DC, USA, 1998. IEEE Computer Society.

[9] E. Kamsties. Understanding ambiguity in requirements engineering. In A. Aurums and C. Wohlin, editors, *Engineering and Managing Software Requirements*, pages 245–266. Springer-Verlag, 2005.

[10] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990. Report CMU/SEI-90-TR-21.

[11] N. Kiyavitskaya, N. Zeni, L. Mich, and D. M. Berry. Requirements for tools for ambiguity identification and measurement in natural language requirements specifications. *Requirements Engineering*, 13(3):207–239, 2008.

[12] G. J. Klir and B. Yuan. *Fuzzy Sets and Fuzzy Logic, Theory and Applications*. Prentice Hall, 1995. Standard Reference for Fuzzy Sets and Fuzzy Logic ISBN: 0-13-101171-5.

[13] K. Lee, K. C. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. In *ICSR-7: Proceedings of the 7th International Conference on Software Reuse*, pages 62–77, London, UK, 2002. Springer-Verlag.

[14] L. Mich and P. N. Inverardi. Requirements analysis using linguistic tools: Results of an on-line survey. In *11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 323–328, 2003.

[15] J. Noppen, P. van den Broek, and M. Aksit. Imperfect requirements in software development. In *Requirements Engineering: Foundation for Software Quality (REFSQ) 2007*, number 4542 in LNCS, pages 247–261. Springer-Verlag, 2007.

[16] A. Pieczynski, S. Robak, and A. Walaszek-Babiszewska. Features with fuzzy probability. *Engineering of Computer-Based Systems, IEEE International Conference on the*, 0:323, 2004.

[17] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[18] S. Robak and A. Pieczynski. Application of fuzzy weighted feature diagrams to model variability in software families. *Artificial Intelligence and Soft Computing*, LNCS 3070/2004:370–375, 2004.

[19] P.-Y. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51:456–479, 2007.

[20] A. Stone and P. Sawyer. Identifying tacit knowledge-based requirements. *IEE Proceedings–Software*, 153(6):211–218, 2006.

Towards End-User Development of Smart Homes by means of Variability Engineering

Francisca Pérez, Carlos Cetina, Pedro Valderas, Joan Fons
Centro de Investigación en Métodos de Producción de Software
Universidad Politécnica de Valencia
Camino de Vera, s/n, 46022 Valencia, Spain
{mperez, ccetina, pvalderas, jfons}@pros.upv.es

Abstract

End users and professional developers possess distinct types of knowledge, end-users know the problem and their needs and developers know the technology to solve the problem. Therefore, it is very important that end-users and designers participate cooperatively. In this paper, we present a design method for designing the adaptation of smart home systems where end-users and technical designers participate cooperatively by means of Variability Engeneering. We introduce Feature Modelling as the underlying technique to enable this cooperation. The paper also shows how to apply two classic end-user techniques within the design method.

1 Introduction

In the past few decades, many research initiatives that are interested in realizing the vision of ubiquitous computing have emerged [26]. These efforts seek to understand how interactive technology can be built into the very fabric of our everyday environment [6]. A growing focus has been placed on transforming the homes we live in into ubiquitous computing environments. In order to achieve the promises of ubiquitous computing, smart homes must improve everyday life activities without losing user acceptance of the system [25]. End-user development copes with this challenge by incorporating user personalization from the very beginning [1, 2].

Smart Homes are complex systems, not only because many different devices are involved, but also because users require these devices to be (1) seamlessly integrated and (2) adapted to their particular needs. Previous studies have highlighted that people continually reconfigure domestic spaces as well as the technologies involved in order to support their activities [24, 23]. The use of end-user development techniques can provide us with several benefits in this aspect. For instance, it provides users with control over an unpredictable combination of interoperating devices [22], and also allows users to customize services at their best convenience [2]. Hence, end-user development pursues a natural alignment between end-user expectations and system capabilities.

It would seem that a complex and adaptive system as a smart home would require sophisticated programming which only skilled software engineers could produce. However, a system properly structured for self-configuring and reactivity to the environment provides exactly the **correct vocabulary** and **control points** for enabling end-users to extend and configure the system. We believe that we can contribute to enabling end-user development using the *Scope, Commonality, and Variability* analysis [10].

In this paper, we present a design method for designing the adaptation of smart home systems where end-users and technical designers participate cooperatively. End-users contribute with their domain knowledge, while designers provide their technical background to preserve the quality of the system. We introduce Feature modelling as the underlying technique to enable this cooperation. This technique not only provides a common terminology to end-users and designers, but also allows for stage configuration through the stepwise specialization of feature models. This is important because both end-users and designers modify system variability at different stages in a cooperative design method. Finally, we show how to apply classic end-user techniques within the proposed design method.

The rest of this paper is structured in the following way: Section 2 analyzes how the system evolves in order to properly fit end-user needs. Section 3 introduces a design method that allows designers to describe adaptation of smart home systems in cooperation with end-users. Section 4 applies classic end-user techniques in different stages within our method. Finally, section 5 concludes the paper.

2 Adaptation in smart home systems and end-users

Smart Home environments are abundant of physical devices such as sensors (i.e. presence or light intensity), actuators (i.e. lamp or alarm) and multimedia (i.e. digiframe or set-top box). The functionality of these devices is augmented by high level *Services*. These Services coordinate the interaction between devices to accomplish specific tasks. For instance, an Automated Illumination service detects presence in a room. To achieve this goal, the Automated Illumination service coordinates lights and presence detectors. In relation with this, by *configuration* of a system we mean the specific set of services and devices that system must support to fit user needs. By *adaptation* we mean a change from one configuration to another to satisfy a change in user needs.

Thus, we start by analyzing the system adaptation level that is needed to allow the system to properly fit end-user needs. This analysis is based on the following ideas:

a. Adaptation in smart homes is driven by a cyclical influence between user needs and system capabilities [8].

b. Adaptation attempts to avoid the mismatch between user needs and system capabilities [14].

c. The adaptation of a system is limited by a point at which so many changes have been performed that we are really talking about another system [17].

According to these ideas, Figure 1 represents the required adaptation level according to the way in which user needs change. The way in which user needs change is represented by a monotonically increasing function *f(t)* because the more the user influences the system, the more the system influences the user (in accordance with **a**). The intersection *p* represents the point at which a new system has to be designed because the needs are out of scope (in accordance with **c**).

We have divided the life cycle of an adaptive system into three zones (in accordance with **b**):

1. **The Inception Phase.** The aim of this phase is to minimize the mismatch between user needs and capabilities. User-centered techniques (sketching or storyboarding) and end-user programming techniques (metaphors or teaching by example) can be used to incorporate the customization of the system from the very beginning. However, we have noticed that there are important factors that are not known until the system is used. For example, during the development of a pervasive meeting room [21], the system was fixed with an initial setting but when end-users made use of the system, they constantly change the configuration because the initial configuration did not satisfy their needs or their needs changed.

   The distance *d* represents the mismatch between end-user needs and initial system capabilities. In this phase, the goal of design methods is to minimize this mismatch. To achieve this goal, design methods can allow for the introduction of some techniques for the end-user to facilitate to specify their needs more accurately.

2. **The Adaptation Phase.** In this phase, adaptations are performed to fit the evolution of end-user needs. These adaptations can be related to environment devices (upgrades or new devices) or with system services (modifications or new services).

   The distance *r* represents the adaptation range of the system. The goal of design methods is to maximize this adaptation range. To achieve this objective, designers should increase the amount of system variation points.

   In this phase, it would be advisable to introduce some techniques for the end-user to upgrade or reconfigure the system at run-time.

3. **The Transition Phase.** This phase starts when the maximum adaptation level has been reached and new needs surpass the system scope.

   The intersection *p* represents the point from which the system can no longer adapt itself to new end-user needs. The purpose of design methods is to delay this point in time. To achieve this objective, designers should adjust the system scope to the end-user needs and provide mechanisms to properly support variations.

   There are no design techniques involved in this phase, since the designers must develop a new system.

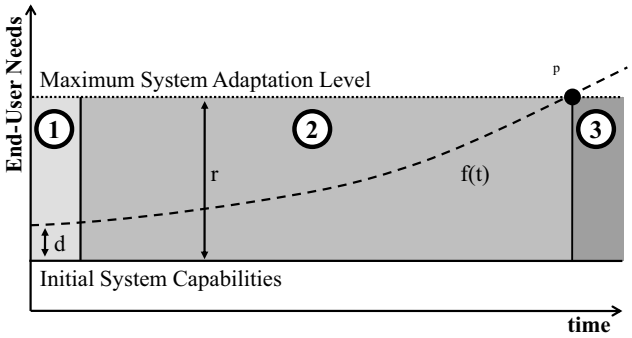


Figure 1. Evolution of end-user needs

In the next section, we present a method for designing adaptation in smart home systems.

3 A model-based method for designing adaptation in smart homes

To define a method to design adaptation in smart homes, we base on the following ideas:

1. **End-users participate in the design process** [18].
2. **System designers cooperate with end-users** [13].
3. **Design encourages customization** [26].

To support these ideas we propose a method based on a feature model. In the next two subsections, we present this feature model and how it is used to describe system adaptation with end-users.

3.1 A feature model for Smart Homes

Many people understand software systems in terms of application features, which are the first recognizable abstractions that characterizes specific systems from the end-user perspective [16]. The feature modeling technique [11] (see Figure 2) is widely used to describe a system configuration and its variants in terms of features. A feature is an increment in system functionality. The features are hierarchically linked in a tree-like structure through variability relationships and are optionally connected by cross-tree constraints. In detail, there are four relationships related to variability concepts on which we are focusing:

- **Optional**. A feature is optional when it can be selected or not whenever its parent feature is selected. Graphically it is represented with a small white circle on top of the feature.
- **Mandatory**. A feature is mandatory when it must be selected whenever its parent feature is selected. Graphically it is represented with a small black circle on top of the feature.
- **Or-relationship**. A set of child features have an or-relationship with their parent feature when one or more child features can be selected simultaneously. Graphically it is represented with a black triangle.
- **Alternative**. A set of child features have an alternative relationship with their parent feature when only one feature can be selected simultaneously. Graphically it is represented with a white triangle.

Additionally, the feature modeling technique incorporates two relationships to express constraints:

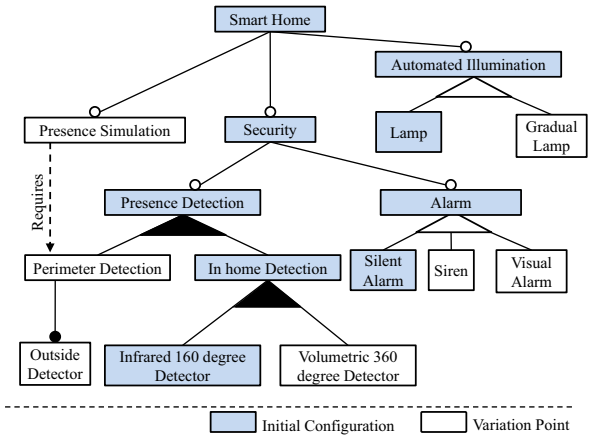


Figure 2. Smart home feature model

- **Requires**. A relationship *A Requires B* means that when A is selected in a system configuration, B is also selected. Graphically it is represented with a dashed arrow.
- **Excludes**. A relationship *A Excludes B* means that when A is selected in a system configuration, B cannot be selected. Graphically it is represented with a dashed double-headed arrow.

We also include **Variation Points** in the feature model. By variation point we mean a feature that has not been selected for the current configuration but can be used to define further configurations. The features filled in gray are the selected features of the smart home configuration, while the white features represent variation points.

An example of a feature model is shown in Figure 2. The feature model describes the smart home with an Automated Illumination and a Security service. This security service relies on presence detection (inside the home) and a silent alarm. Potentially, the system can be upgraded with more services (Perimeter Detection and Presence Simulation) or with more devices (a Siren, a Visual Alarm, a Gradual Lamp or a Volumetric Detector). Note how these potential updates are represented by Variation Points.

For instance, Security Service represented in Figure 2 initially uses a Silent Alarm device. If the system is upgraded with a Siren device the system will also use Siren in the Security Service since the feature Siren is modeled in the Smart Home Feature Model as a Variation Point.

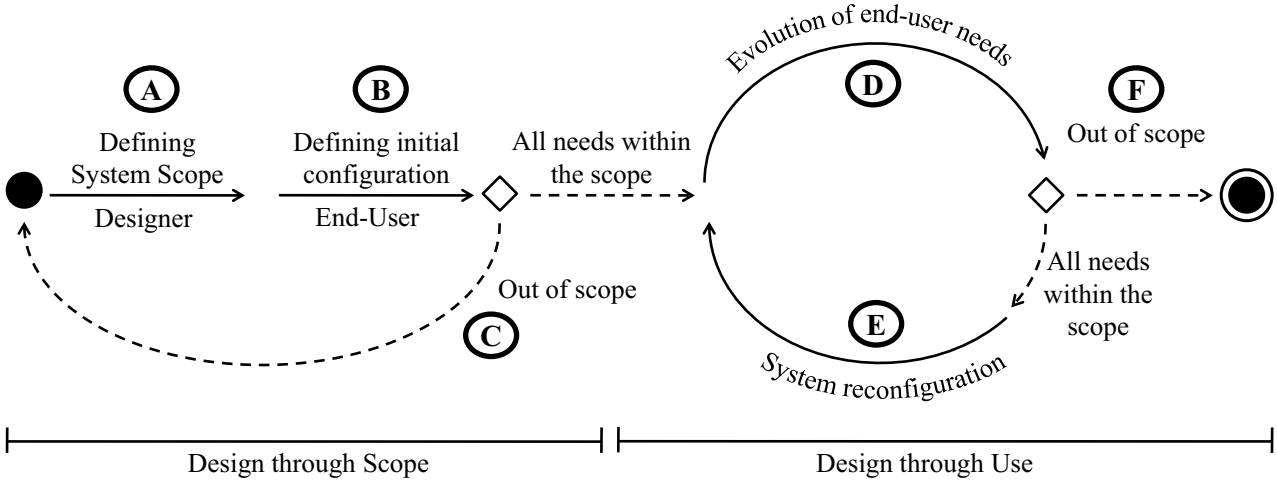


Figure 3. The design method for adaptive smart homes

3.2 A design method for design adapatative smart homes

Our design method is based on our experience with developing smart homes [20, 21, 19]. Figure 3 graphically presents the steps (arrows) and the decisions (diamonds with dashed arrows) of the method. We propose a design process that is divided in two stages: (1) designing through the scope and (2) designing through the use. Both stages are performed by using the feature model presented in the previous section.

- **Design through Scope**. The first stage involves steps A, B, and C (see Figure 3), and its goal is to fit the system scope to the end-user needs. The actors of this stage are the system designer (technological background) and the end-user (domain knowledge). Both cooperate in the design of the system as follows:

  1. **Step A: Defining System Scope**. The designers propose an initial design to a group of target end-users in order to discover where the perceived and realistic needs align. Setting the proper scope of the system is a fundamental step in achieving user acceptance. The scope determines the domain of the smart home (elderly care, kids assistance, security...). The designers set the scope by identifying all the possible features and their relationships in a feature model (see the top left picture of Figure 4).
  2. **Step B: Defining Initial Configuration**.The end-users customize their smart homes from the initial configuration. A configuration is represented as a valid selection of features in the feature model and features which are not selected constitute the variation points (see the top right picture of Figure 4).
  3. **Step C: Out of scope**.The end-users identify needs for which no element exists. Afterwards, this feedback guides designers to focus the system scope on end-user needs by identifying the desired configuration and new variation points (see the bottom right picture of Figure 4).

  This stage is performed again until all needs have been addressed. Rather than forcing the designers to speculate on the system scope, we have found that presenting users with designs that surpass their needs helps to uncover the boundaries of the scope. It is applied in the Inception Phase that is described in Section 2. The aim of this stage is to avoid the initial mismatch between user needs and system capabilities and also to reduce the *rigidity effect* [12] of traditional end-user programming. This effect is introduced by non-cooperative design methods and implies that end-user decisions are restricted by *a priori* specifications.

- **Design through Use**. The second stage involves steps D, E and F (see Figure 3). Its goal is to allow end-users to reconfigure the system in run-time. The actors of this stage are end-users (with their possible new needs) and the system (which must adapt itself to these end-user needs). The variation points identified in the previous stage enables end-users to redesign system performance to match their needs at run-time.The aim of each step is:

4. **Step D-E: Evolution of end-user needs and System reconfiguration**. The adaptation is performed as a transition between sets of features when end-user needs change. Then, the smart home reconfigures itself from a valid selection of features to another valid selection (see the bottom left picture of Figure 4).
5. **Step F: Out of scope**. Users have new needs that are not considered in the scope of the system (represented by the feature model). Incorporating new features to a feature model implies modifying its scope. These modifications imply restarting the design method (see the bottom right picture of Figure 4).

This stage is applied at the Adaptation Phase that is described in Section 2. The aim of this stage is to enable end-users to take control of the system performance, which has been identified as an important design principle [12].

To define these stages, we use the structure presented in the work by Stewart Brand [8], who refers to the successive cyclical influence between users and buildings. He stated:

> "First we shape our building, then they shape us, then we shape them again ad infinitum. Function reforms form, perpetually."

The method presented in this paper describes *which* steps have to be taken. The following section presents some techniques that support end-users in the performance of steps B and D.

4 End-user techniques

End-user design techniques have been adopted from human computer interaction to ubiquitous computing. These techniques encourage and enable end-users to participate in the creation of software. End-users are introduced to the development process by means of appropriate metaphors, such as the jigsaw metaphor [2], the media cubes metaphor [15], the butler metaphor [1] or the fridge magnet metaphor [4]. In the context of this work, these techniques can be used to improve the design of smart home systems adapted to end-user needs. However, they do not address the adaptation of the system when deployed and end-users change their needs. To solve this problem, there are other approaches such as the End-User Sensor Installation Technique [7], Programming by Demonstration [3] or Pervasive Interactive Programming [9].

In this work, we use the Jigsaw metaphor and Programming by demonstration techniques. The next two subsections describe these techniques and how they can be integrated in our cooperative design method. In particular, we show how these techniques can exploit feature models and how they can be used to support Steps B and D where the actor is the end-user.

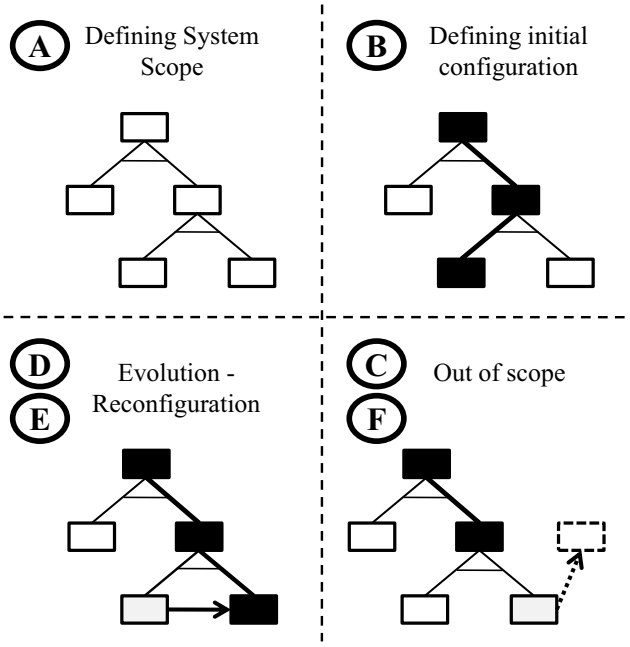


Figure 4. Steps of the design method

4.1 Applying Jisgaw technique in Step B

With the aim to allow end-users to define the initial configuration in Step B, we have chosen the **jigsaw metaphor** [2]. The "jigsaw pieces" metaphor is based on the familiarity evoked by the notion and the intuitive suggestion of assembly by connecting pieces together. Essentially, it allows users to take varibility decisions through a series of left-to-right couplings of pieces. Constraining connections in a left to right fashion also provides users with the sense of a pipeline of information flow.

We can use the jigsaw metaphor to allow end-users to describe the initial configuration. To achieve this, each feature defined in the feature model is presented to end-users as jisgaw pieces. End-users must join these jigsaw pieces to achieve their initial configuration. End-users create a line of pieces for each desired service included in their initial configuration. Compatibility is a must when joining jigsaw pieces. Compatible and non-compatible pieces are defined by the relationships in the feature model. Two pieces are compatible if their associated features are related by means of a relationship defined in the Feature Model (Optional, Mandatory, Or-relationship or Alternative). In Figure 5, we illustrate the initial state of the jigsaw pieces for the feature model previosly modeled by the designer in Step A

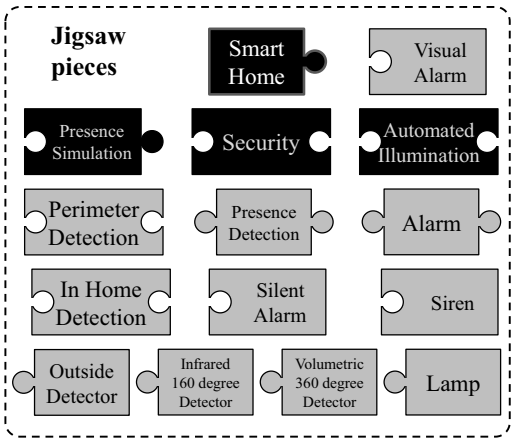


Figure 5. Defining the initial configuration

(see Figure 2). The root piece is filled in black with a gray frame, the compatible pieces are filled in black and the non-compatible pieces are filled in gray. When a jigsaw piece is added, non-compatible pieces are disabled and shadowed, indicating which pieces are compatible.

Therefore, to define an initial configuration end-users must take the following steps:

1. Select the root piece. From this feature end-users can define all their initial configuration services.

2. Add available pieces to the last piece selected. If end-users select a leaf piece a service will be configured.

3. Repeat steps 1 and 2 until all pieces have been selected or repeat until all the services needed are configured.

When the services have been configured, end-users will have a line of puzzle for each service initiated in the system from the root to the leaves. The services which are not initialized will not be available in the system.

According to the feature model and the initial configuration represented in Figure 2, end-users can define three initial services: Presence Detection, Alarm and Automated Illumination. In the end, end-user will attain a line of puzzle for each service (see Figure 6).

4.2 Applying Programming by Demonstration technique in Step D

With the aim to allow end-users customize or upgrade the system at run-time in Step D, we have chosen the Programming by Demonstration (**PbD**) technique [3]. PbD allows "programming" by interacting with the devices in

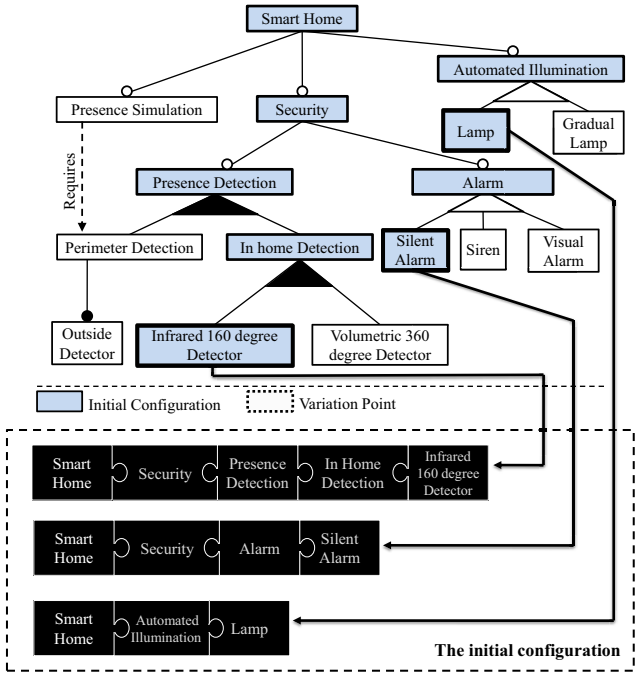


Figure 6. An example of the initial configuration

the environment to show an example of desired functionality. The main advantage of PbD is to allow customizing or upgrading functionality at run-time by end-users who had no experience in developing applications or using complex interfaces. PbD consist of capturing the events which are produced by interacting with the system. Next the system analizes these events.

The way in which PbD can be used in Step D of our method goes as follows. End-users can customize the initial configuration interacting at run-time with devices within a smart home. The communication between end-users and the smart home is via an eventing mechanism. When end-users want to customize a service, they must set the system in **Record mode** to reconfigure this service. Record mode saves the events which are produced when end-users interact with the devices. When end-users stop interacting with the devices they will then disable the record mode. After the record mode is disabled, the system analyses all the saved events and detects which ones are associated with features that belong to the service being reconfigured. If these associations are compatible to the specific service then, the system reconfigures the specific service.

Figure 7 shows an example where the Automated Illumination service is customized. The Automated Illumination service is initially configured to use the Lamp, as we can see in the feature model shown to the left of Figure 7.

Figure 7. Customizing services at run-time

End-users want to customize the Automated Illumination service to switch on the gradual light device rather than the lamp device. To customize this service, end-users must enable the Record mode for this service. Then, end-users go to the sitting room and switch on the gradual lamp. After that, end-users disable the Record mode. When the record mode is disabled, the system checks each saved event. In this example, the system has two saved events: (1) Presence Detection service activated (because the end-user goes into the sitting room) and (2) Gradual lamp is enabled. For each event, the system checks the feature model to search any compatible feature of the selected service. In this case, the event compatible with Automated Illumination service is the activation of the Gradual Lamp. Afterwards, the Automated Illumination service is configured to use the Gradual Lamp, as we can see to the right of Figure 7.

Therefore, end-users can customize or add services in run-time by means of interacting with devices to show the system their desired functionality of a specific service. If the shown funcionality has been modelled in the feature model as having compatible features then, the specific service will be modified and the system will be reconfigured.

5 Conclusion and future work

End-users and professional developers actually possess distinct types of knowledge. End-users are the "owners" of the problem and developers are the "owners" of the technology to solve the problem. End-users do not understand software developers jargon and developers often do not understand end-user jargon [5]. Thus, in this paper we have presented a method which allows end-users to cooperate in the design process of smart home system adaptation and adapt the system to their needs at run-time. We have introduced Feature Modelling as the underlying technique to enable this cooperation. In the design method that we have presented, we have identified two stages: Design through Scope and Design through Use. In Design through Scope, a technical designer proposes an initial feature model setting the proper scope of the system in achieving user acceptance. End-users customize the initial configuration and identify needs for which no elements exists. This stage is performed until all needs are supported. In Design through Use, end-users can upgrade or customize their system at run-time until the system is out of scope needs. Finally, we have shown how two classic end-user techniques can be applied in the context of our method to: (1) allow end-users to cooperate with technical designers and (2) customize or upgrade their system in run-time.

As a future work, we are going to study more end-user techniques to simplify the design system for end-user and find other techniques to be applied with our method. Furthermore, we are going to test more end-user techniques to find the best adaptation in our method. We are also working on the development of editors to apply end-user techniques with our method.

References

[1] *End user empowerment in human centered pervasive computing*, 2002.

[2] *"Playing with the bits": User-configuration of ubiquitous domestic environments*, 2003.

[3] *a CAPpella: programming by demonstration of context-aware applications*, New York, NY, USA, 2004. ACM.

[4] *CAMP: A magnetic poetry interface for end-user programming of capture applications for the home*, 2004.

[5] *End users as unwitting software developers*, New York, NY, USA, 2008. ACM.

[6] G. D. Abowd and E. D. Mynatt. Charting past, present, and future research in ubiquitous computing. *ACM Trans. Comput.-Hum. Interact.*, 7(1):29–58, 2000.

[7] C. Beckmann, S. Consolvo, and A. LaMarca. ome assembly required: Supporting end-user sensor installation in domestic ubiquitous computing environments. *UbiComp 2004*, pages 107–124, 2004.

[8] S. Brand and P. U. S. A. Paper. *How Buildings Learn: What Happens After They're Built*. Penguin Books, October 1995.

[9] Chin, Callaghan, and Clarke. An end-user programming paradigm for pervasive computing applications. *International Conference on Pervasive Services*, 0:325–328, 2006.

[10] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *Software, IEEE*, 15(6):37–45, Nov/Dec 1998.

[11] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. *Software Product Lines*, pages 266–283, 2004.

[12] S. Davidoff, M. K. Lee, C. Yiu, J. Zimmerman, and A. K. Dey. Principles of smart home control. In *UbiComp 2006*, pages 19–34, 2006.

[13] P. Dourish. *Where the Action Is : The Foundations of Embodied Interaction (Bradford Books)*. The MIT Press, September 2004.

[14] Fahrmair, M., Sitou, W., and Spanfelner, B. Unwanted behavior and its impact on adaptive systems in ubiquitous computing. *ABIS 2006: 14th Workshop on Adaptivity and User Modeling in Interactive Systems*, October 2006.

[15] Hague, R., et al. Towards pervasive end-user programming. *UbiComp 2003*, pages 169–170, 2003.

[16] K. Lee, K. C. Kang, W. Chae, and B. W. Choi. Feature-based approach to object-oriented engineering of applications for reuse. *Softw. Pract. Exper.*, 30(9):1025–1046, 2000.

[17] H. Lieberman, F. Paternò, and V. Wulf. *End User Development*. Springer, 2005.

[18] C. Lueg. On the gap between vision and feasibility. In *Pervasive '02: Proceedings of the First International Conference on Pervasive Computing*, pages 45–57, London, UK, 2002. Springer-Verlag.

[19] J. Muñoz and V. Pelechano. Building a software factory for pervasive systems development. In *CAiSE*, pages 342–356, 2005.

[20] J. Muñoz and V. Pelechano. Applying software factories to pervasive systems: A platform specific framework. In *ICEIS (3)*, pages 337–342, 2006.

[21] J. Muñoz, V. Pelechano, and C. Cetina. Implementing a pervasive meeting room: A model driven approach. In *IWUC*, pages 13–20, 2006.

[22] M. W. Newman, J. Z. Sedivy, C. M. Neuwirth, W. K. Edwards, J. I. Hong, S. Izadi, K. Marcelo, and T. F. Smith. Designing for serendipity: supporting end-user configuration of ubiquitous computing environments. In *DIS '02: Proceedings of the 4th conference on Designing interactive systems*, pages 147–156, New York, NY, USA, 2002. ACM.

[23] J. O'Brien, T. Rodden, M. Rouncefield, and J. Hughes. At home with the technology: an ethnographic study of a set-top-box trial. *ACM Trans. Comput.-Hum. Interact.*, 6(3):282–308, 1999.

[24] T. Rodden and S. Benford. The evolution of buildings and implications for the design of ubiquitous domestic environments. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 9–16, New York, NY, USA, 2003. ACM.

[25] A. Schmidt and L. Terrenghi. Methods and guidelines for the design and development of domestic ubiquitous computing applications. *percom*, 00:97–107, 2007.

[26] M. Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):94–104, 1999.

Dealing with Variability in Architecture Descriptions to Support Automotive Product Lines*

Stefan Mann
Fraunhofer Institute for Software and Systems Engineering ISST
Steinplatz 2, 10623 Berlin, Germany
stefan.mann@isst.fraunhofer.de

Georg Rock
PROSTEP IMP GmbH
Dolivostr. 11, 64293 Darmstadt, Germany
georg.rock@prostep.com

Abstract

Architectural description languages (ADLs) are essential means for a system and software design in the large. Their common concepts are components, ports, interfaces and connectors. Some of them already support the representation and management of variance, a prerequisite to support product line engineering, but the support of variability often stops on component level. In this paper, a more detailed view on the integration of variability into architectural models is taken. The focus is set on providing support for product line engineering within the automotive E/E[1] *domain, where functionality and/or its realization is varying according to specific customer needs and hardware topologies. In general, the fundamental question in this domain is not, whether a product line approach should be applied, but what is the best way to integrate it.*

1. Introduction

Architecture description languages (ADLs) are widely used to specify systems and software designs in the large. According to nowadays complexity of embedded software systems in the automotive industry (more than 2000 functions in today's upper class vehicles) architectural models specified in an ADL have to be structured in different layers. We propose to use an appropriate compositional specification formalism that gives us on the one hand the possibility to analyze the described models concerning their variability at each layer separately and on the other hand the possibility to integrate them within a common model in the overall development process. The various layers introduce different perspectives as for example a functional architecture perspective that describes the hierarchical structure and the corresponding interfaces of a system. During the development of complex systems and in a distributed engineering task such a structured and modular approach is indispensable. Besides these aspects it is important to improve the system development process at each stage of such a development, especially with respect to the reuse of certain artifacts within the development process.

In the recent past software product line based development methodologies truly became a mainstream development technique. Although successfully applied in the software development field, product lines are still not state of the art within the automotive domain of embedded software systems. Software and E/E system product lines are build and managed often by explicitly naming and configuring all variants in advance and maintaining all single products through the complete development process.

Product lines have to be engineered within a larger system context of hardware and software systems. The proposed layered approach respects the current development artifacts and processes and introduces variability concepts only where needed. As a reference, the following artifacts depicted in Figure 1 were considered within the VEIA project. Products and their variability are expressed using feature models as they were introduced in [27, 39]. Logical architectures are described using an architecture description language that al-

*This work was partially funded by the Federal Ministry of Education and Research of Germany in the framework of the VEIA project (German acronym for "distributed engineering and integration of automotive product lines") under grant: 01ISF15A. The responsibility for this article lies with the authors. For further information cf. the project's website: http://veia.isst.fraunhofer.de/.

[1] electric / electronic

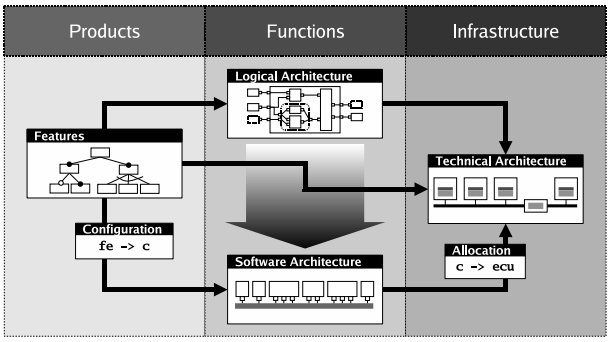


Figure 1. Artifacts of the VEIA reference process.

lows to define variability of different specification elements as there are optional functions, ports or function variation points (see section 2). These logical architectures are directly connected via a *refinement relationship* to the description of so called software architectures representing an implementation of the logical architecture. This relationship is used to trace changes made on different levels and to semantically connect the descriptions on these levels to allow for a simultaneous configuration process. Both the logical architecture and the software architecture are connected to an infrastructure layer via a deployment mapping. This infrastructure layer specifies the topology and hardware restrictions the upper layers rely on. Within the reference process a pervasive staged configuration process is established that starts with the overall feature model and is executed through all the layers. Within this process local constraints arising at each layer are respected and analyzed to detect inconsistencies. The mentioned refinement relations are formally interpreted during the transformation process and used as constraints within the configuration process.

Variability has to be represented in all architecture models. The relationships between the artifacts regarding variability issues stem from the idea of the *variability pyramid* in [33], and accordingly reflect the *sources of variation* [7] on the right level. In more detail, they have to incorporate the concept of binding times for variability [40]. In [19] the mentioned binding time problem was addressed by specifying a function net for the product line of a system functionality in vehicles (see section 4) as well as two concurring software architectures. By the use of product line metrics the two software architectures were compared [18].

The work presented here relies on (integrated) concepts from the areas of ADLs (cf. e.g. [29, 14, 12, 13, 31]), product line engineering [40, 8, 33], and automotive systems engineering (e.g. [11, 35, 9, 37, 3]). The work is based on experiences made in projects on behalf of BMW Group where the introduction of systems and *function orientation* into the engineering processes were focused [20, 21, 28]. Thus, methods to support a distributed engineering and integration of automotive product lines are developed within the BMBF funded research project VEIA, cf. [17, 18, 19, 26, 25, 22, 23, 24, 16, 15]. Our prototypical implementation of a case tool called *v.control* demonstrates different aspects of the sketched methodology.[2]

In this paper we introduce the underlying concepts for the integration of variability modeling aspects into architectural models (see section 2) and a possibility to automatically compute the corresponding feature tree for formal analysis purposes (see section 3). Doing so we are able to analyze the variance of the product line with respect to architectural views and abstraction levels, to assess alternative solution strategies with respect to their variability aspects, and to evaluate them regarding development costs and efforts. We examined the presented method with the help of an example use case described in detail in section 4. The paper ends by mentioning related work (section 5) and some concluding remarks (section 6).

2. Variability concepts in architecture descriptions

Components are the building blocks of an architecture in common ADLs. *Interface descriptions* (*ports*) specify the ability of combining components. *Connectors* establish the composition by connecting ports. The composition of components results in higher-level components again, see e.g. [29]. As an assumption for the following discussion, a higher level component is just a cluster of its subcomponents and does not add any additional behavior or hides anything. Thus, all ports not connected between the subcomponents are delegated to the parent component. The architectural artifacts of the VEIA reference process (cf. Figure 1) are specializations of such a general component model. Because of the automotive domain we concentrate on signal-based communication between components in functional views.

Common variability concepts for product lines, e.g. found in [27, 40, 33, 36], are dealing with *optionality*, *alternatives* (encapsulated by XOR variation points) and *OR variability*. *Parameterization* is also often provided. Furthermore, *dependencies* (e.g. “needs”, “requires” or other logical statements) are used to constrain possible combinations of variable elements.

In order to represent product lines on an architectural level, variability concepts need to be consistently integrated with the above mentioned architectural concepts. The integration, as sketched in the following, results in new kinds of modeling elements on different levels of granularity:

[2] The implementation of the demonstrator is still work in progress. All mentioned methods and analysis operations are prototypically realized within *v.control*.

- Applying the optionality concept on components results in a differentiation of composition, i.e. we get a distinction in *mandatory* and *optional subcomponents*.
- The integration of XOR variability into components results in a new hierarchical relationship between components: A *component variation point* represents a generalization of alternative components as it exhibits their commonalities with respect to the architectural role which the alternatives are able to fulfill.[3]
- By the application of variability concepts on horizontal connectors, we can distinguish between *mandatory* and *optional connectors*. XOR variability (like "switches" in Koala [32]) is not supported, because this situation can be emulated by a variable component fulfilling this task, or by a component variation point.

  Delegation connectors are used to relate two hierarchical levels. They cannot exhibit own variability because of our definition of the composition of components.
- The consistent integration of the variability concepts yields components with varying interfaces in the form of *optional* vs. *mandatory ports*, and in the form of *ports with fixed* or *varying signal sets*.
- Parameterization is applicable on all architectural elements, e.g. *parameterized* components, ports, connectors, or signals, whereby variability is supported for their attributes.

These basic concepts are not isolated, but interrelated. Our approach allows—and thus deals with—the situation when variability *within* a component cannot be fully encapsulated and hidden by that component. Such a situation happens when a component has optional subcomponents or subcomponents with optional ports. Consider for instance the optional output port `poutDisplayEnhanced` of the mandatory component `CbsDisplay` in Figure 5. When this port is available in one product, the corresponding communication is delegated to the environment of the parent component `Cbs`. Another situation when inner variability cannot be hidden is often introduced by component variation points. Alternative components architecturally play the same role, but they are related to different requirements or solutions. This can cause a different demand on signals they send or receive. The comparison of the alternatives with respect to their ports leads to the distinction of ports which are common to all alternatives (e.g. input port `pinKm` of the component variation point `CbsWdSparkPlugs`[4] in Figure 5), and variant-specific ports which are not present in all alternatives (e.g. port `pinSensorsSp` of the same component). The component variation point exhibits the result of this comparison.

Origin of variability	Occurrence	Signal set	Kind of ports wrt. variability	Notation
independent	always	fixed	mandatory, fixed port (*"invariant port"*)	▶
		varying	mandatory, varying port (*"port variation point"*)	▷
	sometimes	fixed	optional, fixed port	▶
		varying	optional, varying port	▷
dependent	always	fixed	*not applicable*	▶
		varying	dependently varying port	▷
	sometimes	fixed	dependently optional, fixed port	▶
		varying	dependently optional, varying port	▷

Table 1. Variability of ports.

In general, we provide a minimal / maximal view of the communication needs of a component which has inner variability. The minimal view only comprises the common elements (the invariants). In contrast, the maximal view comprises the complete set of possible elements (i.e. invariant as well as variant-specific elements). Furthermore, it is distinguished whether the variability of the element originates from another element, i.e. if it's independent or dependent. Dependency is established along the hierarchical structure of components, from lower to higher components. Ports have to represent this differentiation, fully characterized by the following three variability criteria:

1. *Occurrence of a port:* A port can be a mandatory or an optional "feature" of a component, i.e. a port is distinguished whether it *always* (part of all products) or *sometimes* (in some products of the product line) occurs.
2. *Signal set of a port:* The information items communicated via a port can be *fixed* or *varying*. A port can have different signal sets because of the introduction of alternative components. The alternatives could exhibit that they all need information from a specific component, but they differ in the specific signal set. In this case, the

[3] We do not focus on an explicit support of OR variability (i.e. selecting any number of elements of a set) in architectures, although it can be a useful construct. In case that OR variance is incorporated additional constraints on the underlying levels that describe not only the architecture but also the behavior of different development artifacts have to be incorporated. These constraints are concerned with communication or synchronization issues that are inherited from the variance modeling at the upper layer.

[4] A component variation point is represented by a rounded rectangle in our graphical notation, its alternatives are the direct subcomponents. Dashed rectangles mark components as optional.

Component	Port occurrence	Port's signal set	Possible kinds of ports on the component	
Atomic component	always	fixed	invariant port	▶
	sometimes	fixed	optional, fixed port	▷
Hierarchical composed component	always	fixed	invariant port	▶
		varying	dependently varying port	◑
	sometimes	fixed	dependently optional, fixed port	▷
		varying	dependently optional, varying port	◑
Component variation point	always	fixed	invariant port	▶
		varying	port variation point	◑
	sometimes	fixed	optional, fixed port	▷
			dependently optional, fixed port	▷
		varying	optional, varying port	◑
			dependently optional, varying port	◑

Table 2. Components and their ports wrt. to variability.

corresponding port at the component variation point is represented as a varying port.

3. *Origin of variability of a port:* The variability can originate from lower level components, thus a port can be *independent* or *dependent* with respect to its occurrence or its signal set. Reconsider the output port `poutDisplayEnhanced` of function `CbsDisplay` in Figure 5. The corresponding port of function `Cbs` is dependent. The same effect applies to an delegated mandatory port of an optional subcomponent (e.g. port `pinSensorsPf` of the optional component `CbsWdParticleFilter`). In general, there is architectural independence of ports at atomic components, because their inner variability is not represented on the architectural level.

The combination of the three variability criteria results in different kinds of ports as summarized in Table 1. Mandatory, fixed ports represent ports already known from common architectural models, where no variability concept is explicitly integrated. Because mandatory, fixed ports are always present (i.e. invariant), when its component is present, they need not be configured. Thus, there is no dependent version of them.

Along the hierarchical composition relation between components, the variability of a component is propagated to its

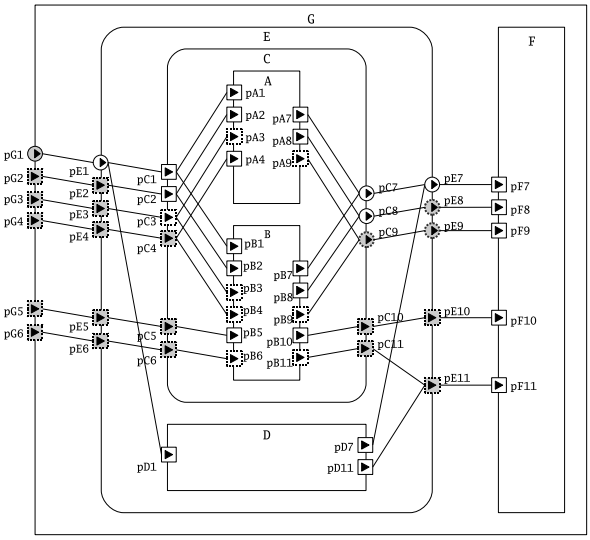


Figure 2. Port dependency because of component variation points.

upper level components. The possible combinations between components and ports are listed in Table 2. How variability caused by alternative components is propagated to upper levels is illustrated in Figure 2. Consider, for instance, how mandatory or optional ports of the alternative components `A` and `B`, which are encapsulated by the component variation point `C`, are delegated to the upper levels. The invariant port `pC1` states that the alternatives `A` and `B` share a common port with the same signal set. In contrast, the varying port `pC7` states, that both components have a common port (with respect to its architectural role), but with different signal sets.

The integration of variability concepts also introduces the need to consistently configure the variable elements occurring within an architecture description. The notions *dependent* and *independent* as above introduced with respect to ports, represent an additional specification means to classify occurring variability. The information whether a specification element is dependent can be automatically computed, thus supporting the user for example by determining the mininal set of configuration points to generate a complete configuration.[5]

To which extend the described concepts are utilized during the analysis and how far the implementation of the mentioned concepts is realized within the demonstrator *v.control* of the VEIA project is sketched in the next section.

[5] This feature is currently not realized within v.control.

3. Analysis of feature models

There are a lot of proposals in literature to formally analyze feature models. Most of them translate a feature model into a representation suitable for a formal analysis. Prominent examples are *Feature Modeling Plugin (FMP)* [39], *pure::variants* [34] or the *FAMA Framework* [10].

The analysis engine of *FMP* is based on binary decision diagrams (BDD). *pure::variants* uses the logical programming language Prolog to reason about feature models. These analysis techniques are used to prove certain properties of feature models that were related to the following questions or operations:

1. Does the feature model has at least one valid configuration (satisfiability) and if so, how many models (configurations) are represented by that feature model?
2. Does the feature model contain features that were never part of a valid configuration (dead feature detection)?
3. The operation to derive a (partly) configured product out of a feature model is most important during the development process and strongly connected to the binding of variance.
4. If a property such as satisfiability cannot be established, then the user should not simply get the answer "no", but should get a helpful explanation on the basis of his feature model to be able to analyze the fault and repair it.
5. The ability to prove arbitrary properties about a feature model is concerned with nearly all the before mentioned operations and questions. It gives the user the possibility to formally examine his feature model to ensure the desired feature model behavior on a formal, objectifiable and repeatable basis.

Within the VEIA project we aim at providing a proof of concept in terms of a prototypical implementation of a feature modeling tool that is able to answer all the enumerated questions and is not limited to Horn-formulae for example. The technique that we propose is based on the same idea as the tools mentioned above. We use a transformation approach that translates a given feature model into a *propositional logic* or *first-order logic* formula. This approach allows us to define arbitrary conditions on features that are expressible in the respective logic. These conditions represent constraints on the corresponding feature model that have to be respected if satisfiability is checked or within a configuration process. We decided to use the automatic theorem prover *SPASS* [30] as our reasoning engine. Such a theorem prover is able to formally analyze problems with a large number of features and it can be used to solve the constraints arising during the configuration process. Thus, all the above mentioned questions can be answered using this approach that is completely implemented within *v.control*. We further expect to scale up with large feature models, since many techniques used to implement for example a constraint propagation mechanism are already successfully used as proving strategies within such theorem provers. Nevertheless, the integration into the complete development process that is concerned with different refinement levels (see Figure 1) is still not solved completely. In the following we sketch a proposal how to connect feature models with the underlying architectural models especially with respect to the configuration of feature models.

As mentioned before we propose a layered approach for the separation of concerns on the different levels. Within such a structured approach a connection between the various layers has to be established that maps the feature model to the architectural artifacts. This mapping gives us on the one hand the possibility to trace changes over the complete development process and on the other hand allows for an automatic computation of *model configurations*. In the following we substantiate the notion of configuration models with the help of a simple example taken from the CBS scenario described in section 4.

In our approach we use feature models and the corresponding operations defined on them as the central variability management engine. The integration of this engine into a system development process is one of the major tasks for an enterprise wide consistent and non-isolated variability management approach. To this end we use a translation process that integrates the development artifacts from different layers into one single feature model[6]. Within the computation of this model the variability analysis presented in section 2 is used to formulate the respective constraints on functions, ports and their connections.

The translation algorithm is based on translation rules that constitute the mapping from architectural elements to feature model elements (propositional formulas) as illustrated by the following selection of rules.[7]

R1 An atomic function F may have mandatory ports $P_1, \ldots, P_n$ which are part of a configuration if and only if the function itself is part of that configuration, expressed by: $F \Leftrightarrow (P_1 \wedge \ldots \wedge P_n)$.

R2 An atomic function F may have optional ports $P_1, \ldots, P_n$. If one of the optional ports is part of a configuration, then the function F is part of that configuration, expressed by: $(P_1 \vee \ldots \vee P_n) \Rightarrow F$. Note that in this case the translation excludes configurations where ports exist with no associated function.

[6] used as an internal computation and analysis model

[7] The complete set of rules realized within *v.control* covers all syntactical possibilities used during the specification of a functional architecture.

R3 A hierarchical (non-atomic) function F is decomposed into sub-functions $F_1, \ldots, F_n$. These sub-functions may be mandatory, i.e. if their parent is present, then the sub-functions are present, too. A sub-function can be an atomic function, a hierarchical function, or a function variation point. The corresponding translation is given by the following formula: $\forall\, i : 1 \leq i \leq n : F \Leftrightarrow F_i$

R4 As described in R3 a hierarchical function F can be decomposed into sub-functions $F_1, \ldots, F_n$. These sub-functions may be optional. Thus, their presence within a configuration depends on the presence of their parent function and on the fact whether they are selected during that configuration. The simple formal translation is given by the following formula: $\forall\, i : 1 \leq i \leq n : F_i \Rightarrow F$

R5 A *variation point* is a function F which encapsulates the logical *XOR*-relation between its sub-functions $F_1, \ldots, F_n$. If a function variation point is present then exactly one of its sub-functions is present.[8] Which of the sub-functions $F_1, \ldots, F_n$ is taken depends on the configuration of the function variation point. The alternatives may again be a function variation point, or an atomic or hierarchical function. The formal translation is reflected by the following formula:

$$
\begin{array}{l}
(\forall\, i : 1 \leq i \leq n : F_i \Rightarrow F) \\
\wedge \\
(F \Rightarrow (F_1 \vee \ldots \vee F_n)) \\
\wedge \\
(\forall\, i,j : 1 \leq i,j \leq n : i \neq j : F_i \Rightarrow \neg\, F_j)
\end{array}
$$

The given rules can be applied recursively to architectural specifications resulting in a feature tree with the corresponding constraints that at first represents the variability occurring within the functional architecture. Note that this representation does not represent the functional architecture itself. It simply exploits the architecture description elements in order to unambiguously represent their variability. The given formal representation can then be used to formally reason about the variability and to prove for example whether a configuration is consistent.

To illustrate the mentioned approach let us assume we have finished the description of the feature model that represents the product line description in Figure 1 and is shown in Figure 4. Within the scenario described in section 4 we have specified the corresponding function net as illustrated in Figure 5. The function `CbsWdSparkPlugs` is described as a variation point introducing two alternatives. These alternatives are mirrored within the corresponding feature tree. Let us assume for illustration issues that the alternative within the feature tree between adaptive and linear computing has

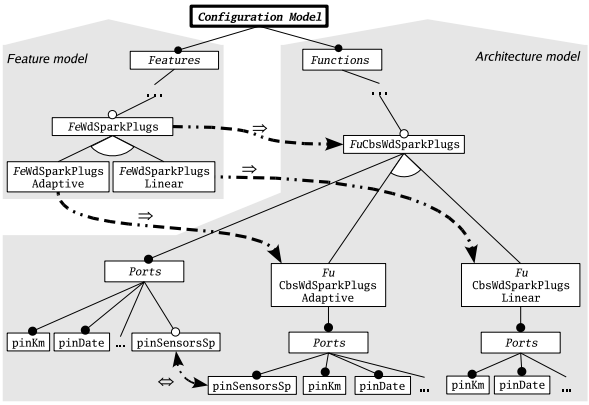


Figure 3. Extended feature tree for analysis and configuration.

not been made before (so we cut the feature tree after the `WdSparkPlugs` node). This leads us to the problem of introducing a new variability within the function net that is not reflected within the feature tree. Although it is possible in such a situation to let the user repair this model inconsistency, we think that such an approach is error prone and it should be possible to automatically compute a new feature tree that incorporates the newly introduced variability. In our example we assume that the feature node `WdSparkPlugs` is mapped to the function `CbsWdSparkPlugs`. From this we can conclude that there is a new variability since the `CbsWdSparkPlugs` represents a variation point. Now the computation is easy as illustrated in Figure 3. The general idea is to represent functions as features and their sub-functions as children of these features. The ports associated to a function are collected under a feature `Ports`. The dependencies between ports[9] are expressed using needs-links as depicted in Figure 3. For the sake of readability Figure 3 shows only one needs link between the port `pinSensorsSp` of the function `CbsWdSparkPlugsAdaptive` and the port `pinSensorsSp` of the function `CbsWdSparkPlugs`. The connection between the mentioned ports is only a delegating connection which is also expressed by the established equivalence relation. For the underlying development artifacts, which have to be connected to the corresponding features in the feature tree, it means that both ports can be identified. The complete list of ports can in general be avoided or hidden since most of them are mandatory and not part of some variability. Besides this we suggest to introduce a general hiding concept that lets the user choose the view (level of detail) of the feature tree.

Based on this automatically computed feature tree the user is able to configure its development artifacts on each

[8] This holds for a completely configured product.

[9] occurring as delegation or horizontal connectors

layer of the development process with the help of the central feature tree (see Figure 7). What remains to be introduced is a concept of artifact dependencies that can be established between different layers in the central feature tree.

4. Case study

The presented concepts are evaluated by a case study from our project partner BMW about a distributed supporting functionality in vehicles: "Condition-based service" (CBS) [19]. Hereby, necessary inspections are not any longer terminated at regular and fixed durations of time or driven distances, but on the real wearout of relevant wearing parts of the vehicle like motor oil quality, brakes, spark-plugs etc. Variability within CBS is primarily caused by different business service concepts for specific vehicle types, different infrastructures of supported vehicle types, and by organizational and development process-related aspects. It is reflected in a varying information basis for CBS, different algorithms to estimate the wearout, and different solution strategies to implement the functionality.

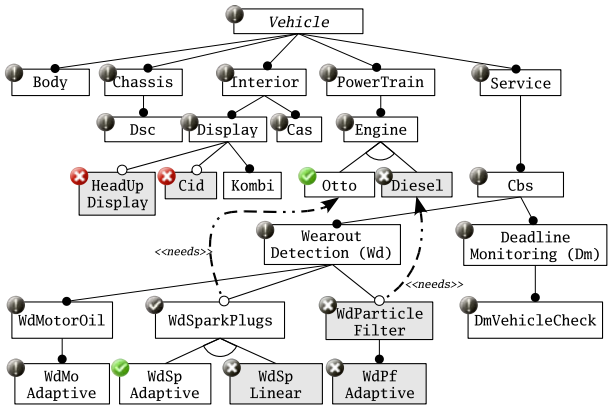


Figure 4. CBS product line description by features (incl. configuration for a product).

The definition of a (simplified) CBS product line is shown in Figure 4. This feature model was configured for a typical product: a vehicle with Otto engine, no additional displays, but where the wearout of spark plugs is detected by an adaptive computing method. The corresponding function net is shown in Figure 5. Each wearing part is represented by a function to compute the wearout. Since the product line supports two ways of computing the wearout of spark plugs, the alternative functions `CbsWdSparkPlugsLinear` and `CbsWdSparkPlugsAdaptive` are introduced. The adaptive variant needs additional input from the spark plugs sensors for the computation. This is represented by an additional, variant-specific port at the `CbsWdSparkPlugsAdaptive` function,

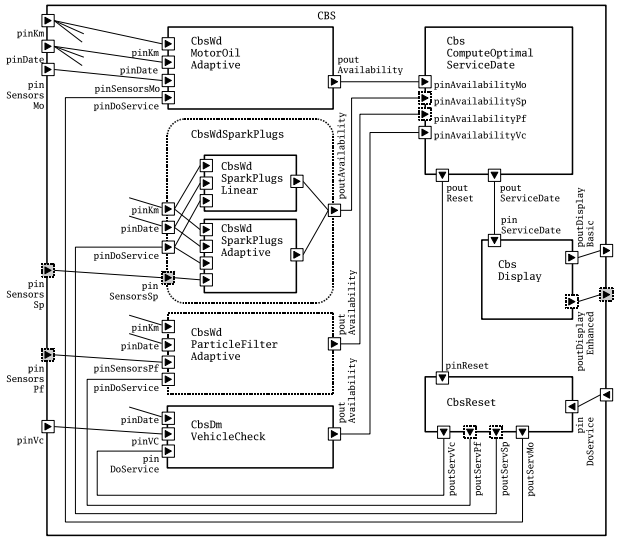


Figure 5. Function net for CBS.

which becomes a dependent port at the functional variation point `CbsWdSparkPlugs` as well as at the (top-level) function `Cbs` (port `pinSensorsSp`). Furthermore, the wearout detection function for spark plugs as well as for the particle filter are optional because of hardware dependence: vehicles equipped with an Otto engine have spark plugs only, and vehicles with Diesel engines have particle filters only. Both wearout detection functions could have been modeled as alternatives in the function net, but we've decided to model them just as options, because they do not represent "functional" alternatives. Generally, the definition of XOR variation points or the usage of options is always a design decision dependent on specific objectives and situations.

5. Related work

The work presented here is related to a number of different efforts. Various ADLs in the literature incorporate the hierarchical composition of components, cf. [29] for an overview and a taxonomy for comparison of ADLs. ADLs are also object of research for automotive or embedded systems like [11, 12, 32, 14, 5, 6].

Variability management and product line engineering is in the scope of many efforts. ADLs for product lines often introduce structural variability for components, but only deal with constant, maybe optional, interfaces for components. A comparison of those ADLs is given in [1, 36]. The notation we used for ports was inspired by [32]. In contrast to our approach, this ADL supports alternative components by alternative connections ("switches"). By "diversity interfaces" a hierarchical parameterization of components and its elements is supported.

An overview about binding times of variability and variability mechanisms is given in [40, 7, 38, 8]. General notions about commonality and differences are published in e.g. [27, 33]. Exploiting feature models for variability management is done by e.g. [39, 34]. pure::variants [34], a configurator tool according to [8], uses a generic family model as abstraction of blocks of a target language description (e.g. source code). By configuration of a feature model, the family model is used to concatenate the blocks to build a product. In this way, our architectural models can be regarded as specializations of the family model. Thus, our concepts could be integrated with such tools.

6. Conclusion

The presented method introduces the possibility to connect different layers within a structured development process in a generic way resulting in an effective approach to configure and to compute products according to predefined measures. It supports the user in finding valid configurations and guarantees that the constraints are not violated. The proposal is flexible in the sense that it allows to incorporate variance that is introduced at later stages of the development without changing the before specified development artifacts. Those artifacts that are specified and defined below the level of the functional architecture can be integrated analogously resulting in a pervasive development of variability throughout the complete product development process.

Although the concepts for the handling of variability are not yet stable within the AUTOSAR considerations [2], our method provides a mean to realize a pervasive handling of variability throughout a product development process that starts with the requirements specification and ends with an AUTOSAR compliant software development [3, 4].

As proof of concept the work is prototypically implemented in a case tool which we called *v.control*. It demonstrates different aspects of our methodology: the specification of product lines in form of function and software component architectures including abstraction and reuse issues, the assessment of a product line by the evaluation of the architecture specification using metrics, and the configuration of a product line architecture in order to consistently derive the specifications of each of its products. For the latter we use common feature modeling techniques for a centralized and continuous variability management. The screenshots of the VEIA demonstrator *v.control* in Figure 6 and Figure 7 illustrate the implemented functionality with respect to the method presented in this paper. The screenshot depicted in Figure 6 shows how the linkage between features and functions is presented to the user. A simultaneous configuration of both the feature model and the connected functional architecture is shown in Figure 7.

The concepts realized within the demonstrator have to be completed in the future with respect to data management issues as for example the change management of configurations, feature models and function models. Furthermore, it is planned to allow for more flexibility within the current fixed three tier development methodology.

References

[1] T. Asikainen. *Modelling Methods for Managing Variability of Configurable Software Product Families*. Licentiate thesis of science in technology, Helsinki University of Technology, 2004.

[2] AUTOSAR Development Partnership. AUTOSAR – AUTomotive Open System ARchitecture. `http://www.autosar.org/`.

[3] AUTOSAR Development Partnership. AUTOSAR Methodology. Release 3.1, doc. 068, version 1.2.2, 2008.

[4] AUTOSAR Development Partnership. AUTOSAR Technical Overview. Release 3.1, doc. 067, version 2.2.2, 2008.

[5] AUTOSAR Development Partnership. Software Component Template. Release 3.1, doc. 062, version 3.1.0, 2008.

[6] AUTOSAR Development Partnership. Specification of the System Template. Release 3.1, doc. 063, version 3.0.4, 2008.

[7] F. Bachmann and L. Bass. Managing variability in software architectures. In *Proc. 2001 Symposium on Software Reusability (SSR)*, 2001. May 18-20, Toronto, Canada.

[8] F. Bachmann and C. Clements, P.˙ Variability in software product lines. Technical Report CMU/SEI-2005-TR-012, ADA450337, CMU-SEI, Sept. 2005.

[9] A. Bauer, M. Broy, J. Romberg, B. Schätz, P. Braun, U. Freund, N. Mata, R. Sandner, and D. Ziegenbein. Automode – notations, methods, and tools for model-based development of automotive software. In *Proc. of the SAE 2005 World Congress, Detroit, MI, 2005*. Society of Automotive Engineers, 2005.

[10] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortes. FAMA: Tooling a framework for the automated analysis of feature models. In *Workshop on Variability Modeling of Software-intensive Systems (VAMOS 2007)*, Limerick, Ireland, Jan. 16-18, 2007.

[11] P. Braun, M. v. d. Beeck, U. Freund, and M. Rappl. Architecture centric modeling of automotive control software. In *Proc. SAE 2003 World Congress of Automotive Engineers*, SAE Transactions 2003-01-0856, Mar. 2003. Detroit, USA.

[12] EAST-EEA Consortium. EAST-EAA embedded electronic architecture – definition of language for automotive embedded electronic architecture. Project report Deliverable D3.6, Version 1.02, June 2004.

[13] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity–the Ptolemy approach. *Proc. of the IEEE*, 91(1):127–144, Jan. 2003.

[14] P. Feiler, D. Gluch, and J. Hudak. The architecture analysis & design language (AADL): An introduction. CMU/SEI-2006-TN-011, CMU-SEI, Feb. 2006.

[15] M. Große-Rhode. Architekturzentriertes Variantenmanagement für eingebettete Systeme – Ergebnisse des Projekts

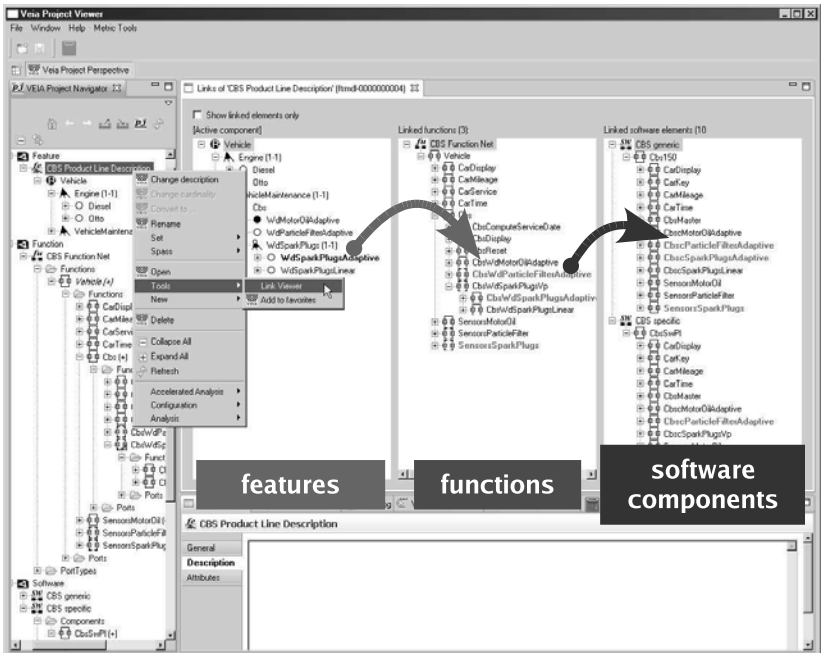


Figure 6. Screenshot of the VEIA prototype "v.control" wrt. configuration: Link viewer

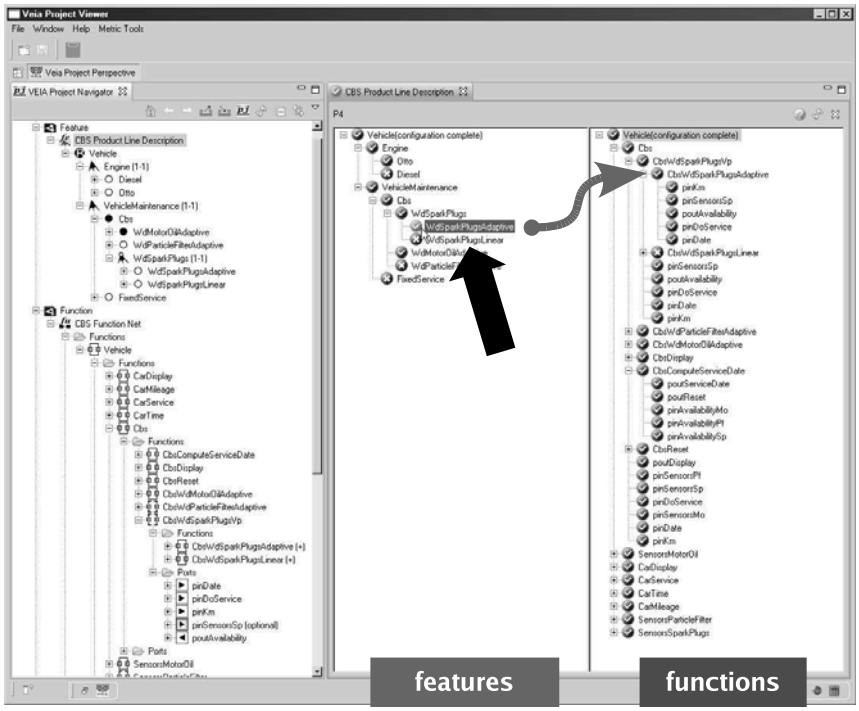


Figure 7. Screenshot of the VEIA prototype "v.control" wrt. configuration: Configuration view.

„Verteilte Entwicklung und Integration von Automotive-Produktlinien“. ISST-Bericht 89/08, Fraunhofer ISST Berlin, Oct. 2008. (in German).

[16] M. Große-Rhode. Methods for the development of architecture models in the VEIA reference process. ISST-Bericht 85/08, Fraunhofer ISST Berlin, May 2008.

[17] M. Große-Rhode, S. Euringer, E. Kleinod, and S. Mann. Rough draft of VEIA reference process. ISST-Bericht 80/07, Fraunhofer ISST Berlin, Jan. 2007.

[18] M. Große-Rhode, E. Kleinod, and S. Mann. Entscheidungsgrundlagen für die Entwicklung von Softwareproduktlinien. ISST-Bericht 83/07, Fraunhofer ISST Berlin, Oct. 2007. (in German).

[19] M. Große-Rhode, E. Kleinod, and S. Mann. Fallstudie „Condition-Based Service“: Modell für die Bewertung von logischen Architekturen und Softwarearchitekturen. ISST-Bericht 84/07, Fraunhofer ISST Berlin, Oct. 2007. (in German).

[20] M. Große-Rhode and S. Mann. Model-based development and integration of embedded components: an experience report from an industry project. In *Proc. 1st Int. Workshop Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2004), Satellite event of ETAPS 2004, April 3, 2004, Barcelona, Catalonia, Spain*, pages 112–117, Apr. 2004.

[21] M. Große-Rhode and S. Mann. Model-based systems engineering in the automobile industry: Positions and experiences. In *Proc. Int. Workshop on Solutions for Automotive Software Architectures: Open Standards, Reference Architectures, Product Line Architectures, and Architecture Definition Languages*, Aug. 2004. Workshop at 3rd Software Product Lines Conference SPLC 2004, Boston, Massachusetts, USA, 30 August – 2 September 2004.

[22] A. Gruler, A. Harhurin, and J. Hartmann. Modeling the functionality of multi-functional software systems. In *Proc. 14th Annual IEEE Int. Conf. on the Engineering of Computer Based Systems (ECBS)*, Mar. 2007.

[23] A. Gruler, M. Leucker, and K. Scheidemann. Calculating and modeling common parts of software product lines. In *Proc. of the 12th Int. Software Product Line Conf.* IEEE, 2008.

[24] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and model checking software product lines. In *Proc. of the 10th IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems (FMOODS08)*, LNCS, 2008.

[25] A. Harhurin and J. Hartmann. Service-oriented commonality analysis across existing systems. In *Proc. 12th Int. Software Product Line Conf.* IEEE Computer Society, Sept. 2008.

[26] A. Harhurin and J. Hartmann. Towards consistent specifications of product families. In *Proc. 15th Int. Symposium on Formal Methods*, volume 5014 of *LNCS*. Springer-Verlag, May 2008.

[27] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) – feasibility study. CMU/SEI-90-TR-21, CMU-SEI, Nov. 1990.

[28] E. Kleinod. Modellbasierte Systementwicklung in der Automobilindustrie – Das MOSES-Projekt. ISST-Bericht 77/06, Fraunhofer ISST Berlin, Apr. 2006. (in German).

[29] N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software-Engineering*, 26(1):70–93, Jan. 2000.

[30] Max-Planck-Institut Informatik, Saarbrücken. SPASS: An automated theorem prover for first-order logic with equality. Theorem prover. `http://spass.mpi-sb.mpg.de/`.

[31] Object Management Group. OMG Systems Modeling Language (SysML) Specification. OMG Document formal/2008-11-01, Nov. 2008. Version 1.1.

[32] R. Ommering. *Building Product Populations with Software Components*. PhD thesis, Rijksuniversiteit Groningen, Groningen, Nederlands, 2004.

[33] K. Pohl, G. Böckle, and F. v. d. Linden. *Software Product Line Engineering – Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[34] pure::systems GmbH. pure::variants. CASE tool. `http://www.pure-systems.com/`.

[35] J. Schäuffele and T. Zurawka. *Automotive Software Engineering*. ATZ-MTZ-Fachbuch. Vieweg, 2003.

[36] M. Sinnema and S. Deelstra. Classifying variability modeling techniques. *Elsevier Journal on Information and Software Technology*, 49(7):717–739, July 2007.

[37] G. Steininger. Functional modelling for architecture design of distributed ecu networks. In *Proc. 3rd Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 3)*, volume 191 of *HNI-Verlagsschriftenreihe*, Paderborn, 2006.

[38] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705–754, Apr. 2005.

[39] University of Waterloo, Canada. Feature modeling plugin (version 0.7.0). CASE tool (Eclipse plugin), 2006. `http://gsd.uwaterloo.ca/projects/fmp-plugin/`.

[40] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proc. 2nd Working IEEE / IFIP Conf. on Software Architecture (WICSA)*, pages 45–54. IEEE Computer Society, 2001.

A Preliminary Comparison of Formal Properties on Orthogonal Variability Model and Feature Models

Fabricia Roos-Frantz *
Universidade Regional do Noroeste
do Estado do Rio grande do Sul (UNIJUÍ)
São Francisco, 501.
Ijuí 98700-000 RS (Brazil)
frfrantz@unijui.edu.br

Abstract

Nowadays, Feature Models (FMs) are one of the most employed modelling language by managing variability in Software Product Lines (SPLs). Another proposed language also in order to managing variability in SPLs is the Orthogonal Variability Model (OVM). Currently, the differences between both languages, FMs and OVM, are not so clear. By considering that a formal language should have a well defined syntax and semantics, some authors had defined syntax and semantics of FMs explicitly. However, in in the definition of OVM, its semantic domain and semantic function are not well discussed. Without this clear definition, we could have a misinterpretation when using OVM diagrams. Our aim in this paper is to clarify and better explore the abstract syntax, the semantic domain and the semantic function of OVM, and to emphasize the differences between FMs and OVM concerning such aspects.

1. Introduction and Motivation

Documenting and managing variability is one of the two key properties characterising Software Product Line Engineering (SPLE) [7]. Over the past few years, several variability modeling techniques have been developed aiming to support variability management [10]. In this paper we take into account two modelling languages: FMs, that are one of the most popular, and OVM. We want to discuss about the differences between both languages.

FM was proposed for the first time in 1990 and currently it is the mostly used language to model the variability in SPL. This model capture features commonalities and variabilities, represents dependencies between features, and determines combinations of features that are allowed and forbidden in the SPL [4].

OVM is a variability model proposed by Klaus Pohl et al. [7] for managing the variability in the applications in terms of requirements, architecture, components and test artifacts. In an OVM only the variability of the product line is documented. In this model a *variation point (VP)* documents a variable item, i.e a system functionality which can vary and a *variant (V)* documents the possible instances of a variable item. Its main purpose are: (1) to capture the VPs, i.e. those items that vary in an SPL, and (2) to represent Vs, i.e. how the variable items can vary and (3) to determine constraints between Vs, between Vs and VPs and between VPs and VPs.

A fundamental concern, when we want to do a reasoning about a language, is to make it a formal language [4]. In the words of Schobbens et al. [9], "formal semantics is the best way to avoid ambiguities and to start building safe automated reasoning tools for a variety of purposes including verification, transformation, and code generation". According to Harel and Rumpe [3], a language is formal when it has a well defined syntax (the notation) and a well defined semantics (the meaning).

Nowadays we have a well defined syntax and semantics to FM languages [9], i.e. we can construct FMs without misinterpretation, because we know what is a correct model and what it means exactly. However, if we are working with OVM we are not sure about the correct meaning of these models and also about the real differences between FMs and OVM. This paper focus on doing a review about OVM's syntax and semantics, which were proposed in the literature, and discuss about the differences between FMs and OVM in order to avoid misunderstanding.

The remainder is organized as follows: Section 2 discusses about the abstract syntax of OVM and compares some of its properties with FMs; Section 3 we comment

*PhD student at the University of Sevilla

about the OVM's semantic domain and FM's semantic domain, and we suggest another semantic domain to OVM; Section 4 we discuss about the OVM's semantic function and FM's semantic function; Section 5 presents our conclusions.

1.1. Feature Models (FMs)

The first feature model was proposed in 1990 [5] as part of the method Feature-Oriented Domain Analysis (FODA). Since then, several extensions of FODA have been proposed. A FM represents graphically a product line by means of combinations of features. A FM is composed of two main elements: features and relationships between them. Features are structured in a tree where one of these features is the root. A common graphical notations is depicted in Figure 1.

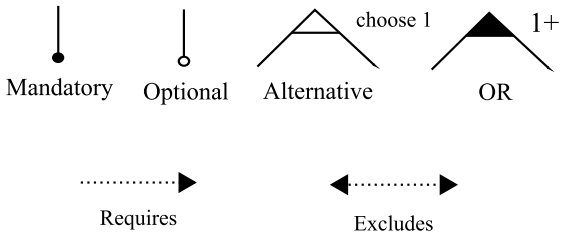


Figure 1: Graphical notation for FM

Figure 2 is an example of feature model inspired by the mobile phone industry. It defines a product line where each product contains two features: *Call* and *Connectivity*. Where *Call* is a mandatory feature and *Connectivity* is an optional feature. It means that all application that belongs to this SPL must have the feature *Call* and can have the feature *Connectivity*. Each product must have at least one of the two types of call, *voice* or *data*, because of the relationship OR. If the product has the feature *Connectivity*, then it must have at least one of the two features, *USB* or *Wifi*.

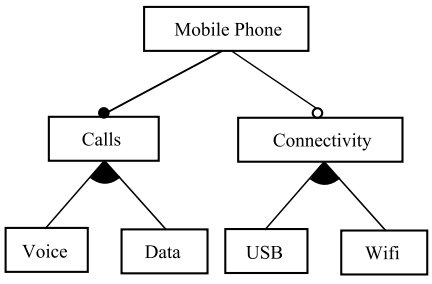


Figure 2: Example of FM

1.2. Orthogonal Variability Model (OVM)

OVM is a proposal for documenting software product line variability [7]. In an OVM only the variability of the product line is documented. In this model a *variation point (VP)* documents a variable item and a *variant (V)* documents the possible instances of a variable item. All VPs are related to at least one V and each V is related to one VP. Both VPs and Vs can be either optional or mandatory (see Figure 3). A mandatory VP must always be bound, i.e, all the product of the product line must have this VP and its Vs must always be chosen. An optional VP does not have to be bound, it may be chosen to a specific product. Always that a VP, mandatory or optional, is bound, its mandatory Vs must be chosen and its optional Vs can, but do not have to be chosen. In OVM, optional variants may be grouped in *alternative choices*. This group is associated to a cardinality *[min...max]* (see Figure 3). Cardinality determines how many Vs may be chosen in an alternative choice, at least *min* and at most *max* Vs of the group. Figure 3 depicts the graphical notation for OVMs [7, 6].

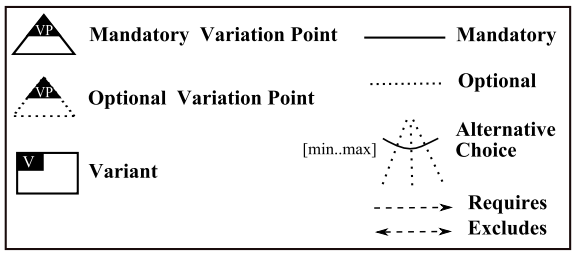


Figure 3: Graphical notation for OVM

In OVM, constraints between nodes are defined graphically. A constrain may be defined between Vs, VPs and Vs and VPs and may be an *excludes* constraint or a *requires* constraint. The excludes constraint specifies a mutual exclusion, for instance, a variant *excludes* a optional VP means that if the variant is chosen to a specific product the VP must not be bound, and vice versa. A *requires* constraint specifies an implication, for instance, a variant *requires* a optional VP means that always the variant is part of a product, the optional VP must be also part of that product. Figure 4 depicts a example of an OVM inspired by the mobile phone industry.

2. Syntax: abstract and concrete syntax

In graphical languages, such as FMs and OVM, the physical representation of the data is known as concrete syntax. In other words, what the user see, like arrows and squares, is only the concrete syntax. Defining rigid syntactics rules

	Nodes	Hierarchical structure	Multiple inheritance	Variation Points	Complex constraints
FM (Batory [1])	Features	yes	no	not explicit	yes
OVM-KP (Klaus Pohl et al. [7])	VPs and Vs	no	yes	Mandatory	no
OVM-M (Metzger et al. [6])	VPs and Vs	no	no	Mandatory / Optional	no

Table 1: Summary of abstract syntax properties.

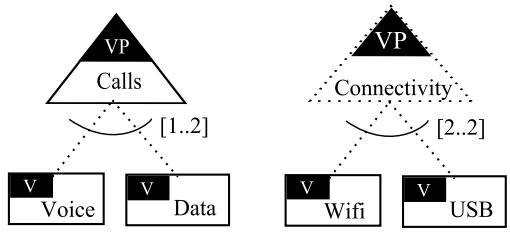


Figure 4: OVM example: mobile phone product line

in visual languages is a difficult task, for this reason, a common practice is to define a formal semantics of the language based on its abstract syntax. The abstract syntax is a representation of data that is independent of its physical representation and of the machine-internal structures and encodings [4].

The first OVM's abstract syntax was proposed by Klaus Pohl et al. [7]. The authors proposed a metamodel, which describes what is a well-formed diagram. Later, Metzger et al. [6] proposed an OVM's abstract syntax which use a mathematical notation to describe a well-formed diagram. In this section we will compare both abstract syntax, underlining the main differences between them. At the same time, we will compare the properties of OVM language with FMs languages according to the abstract syntax. In order to compare both languages we will use the FM proposed by Batory [1], and we will refer to FMs as the proposed in [1].

Table 1 compares some properties about different abstract syntaxes. Each row of this table represents an abstract syntax proposed in the literature. The first one is FM-Batory, which was proposed in [1]. The second is OVM-KP, which was proposed in [7] and the third is OVM-M, proposed in [6]. Each column represents a property of the language. Bellow we describe and comment each one of these properties.

- *Nodes*. We use the term "nodes" to say what a node represents in a graph. For example, in a FM the nodes of the graph are features. It means that each node represents an increment in system functionality. On the other hand, either in OVM-KP or OVM-M, the nodes are VPs (variation points) and Vs (variants), i.e. each functionality of the system that vary is represented by a VP, and each V represents how the VP can vary.

- *Hierarchical structure*. This property states if a graph has a hierarchical structure or not. The FM is represented by a tree and there is one node that is a root. Each node of the tree, with the exception of the root, have one parent. On the other hand, OVM diagrams do not have a hierarchical structure. This diagrams are composed for variation points, which always have child variants. In this diagram there is no a root node, the diagram is composed of a set of VPs with its possible variants.

- *Multiple inheritance*. Happens when a well formed diagram allows a node to have two different parents. The FM does not allow a feature to have more than one parent. When dealing with OVM, this property is defined in two different ways. In Pohl's abstract syntax the diagram can have variants with different VP parents; however, in the second proposal, a variant can have only one VP parent.

- *Variation Points*. Here we consider if graph nodes represent variation points explicitly. In FM all nodes are features, there is no explicit way to represent variation points. The way that FMs represent the variation points, identified in requirements, is through optional or alternative feature. On the other hand, in OVM, all variable item in an SPL is represented by a specific node called VP. In OVM-KP a VP only can be mandatory, i.e. all products of an SPL share this VP. However, in OVM-M, a VP can be mandatory or optional, i.e. if it is mandatory it will be in all products of the SPL; otherwise if it is optional, it will be only in those products which such VP was bound.

- *Complex constraints*. In both languages, FM and OVM, in addition to diagrams there are constraints that restrict the possible combinations of nodes. In FM we can specify constraints more complex than only excludes and requires. For example, we can write constraints like: (*F requires A or B or C*), this means F needs features A, B, or C or any combination thereof.

In OVM, only constraints of type excludes and requires can be specified.

3. Semantics: semantic domain

According to Harel and Rumpe a language's semantics "must provide the meaning of each expression and that meaning must be an element in some well defined and well-understood domain" [3]. In this definition we have two important information about the definition of semantics. First of all the semantics must give a meaning to each element defined in the language's syntax. Second, to define such meaning we need a well defined and well-understood domain. This domain, called "semantic domain" is an abstraction of reality, in such a way that determine what the language should express.

When we were reviewing the literature we realized that the OVM models are treated like FMs, namely, they represent the same domain. But, what means to represent the same domain? Have they the same semantic domain? According to Batory [1] a FM represents a set of products, and each product is seen as a combination of features. Then, a FM represents all the products of the SPL. By considering this, the semantic domain of a FM is considered a product line [9], i.e. the set of sets of combinations of features $\mathcal{PP}(f)$.

The semantic domain of OVM is also considered a product line [6]. Hence, product line is a set of products and each product is a combination of VPs and Vs, then the semantic domain of OVM is a set of sets of combinations of VPs and Vs, i.e. the $\mathcal{PP}(VP \cup Vs)$.

Until now, the semantic domain of OVM has been considered like in FM, the set of sets of combinations of nodes. But, if in OVM we were interested only in variations, we can consider that the semantic domain of OVM is a set of sets of combinations of only variants. Then, the semantic domain of OVM is a product line and each product is a set of variants, i.e. the set of sets of combinations of variants $\mathcal{PP}(V)$. In this way we consider that each product of the product line has only variants and not variation points. For example, for the model of the Figure 4 the semantic domain is the $\mathcal{PP}(V)$, where V is the set {Voice, Data, Wifi, USB}.

4. Semantics: semantic function

The definition of semantics, proposed by Harel and Rumpe, stated that it must provide a meaning of each expression and that meaning must be an element in some domain. The domain is the semantics domain ($\mathcal{S}$) and the expressions are represented by the syntactic domain ($\mathcal{L}$). According to Heymans et al. [4], the set of all data that comply with a given abstract syntax is called the syntactic domain. The function that relates ($\mathcal{L}$) and ($\mathcal{S}$) by giving a meaning to each expression is called semantic function ($\mathcal{M}$). Then, $\mathcal{M} : \mathcal{L} \rightarrow \mathcal{S}$. To every diagram of $\mathcal{L}$, the function $\mathcal{M}$ assigns a product line in $\mathcal{S}$.

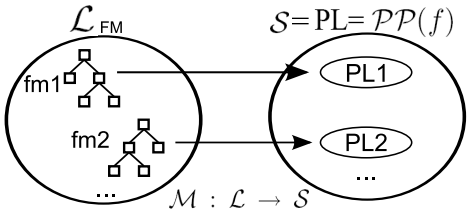


Figure 5: Semantic function of FM

Figure 5 gives an illustration of the FM's semantic function. In this figure we have two different FMs that comply with a FM's abstract syntax, and we have a semantic function that assigns to each diagram a different product line in the semantic domain $\mathcal{M}_F : \mathcal{L}_F \rightarrow PL$ where $PL = \mathcal{PP}(f)$, i.e the power set of set of features. For example, if we have those two diagrams of the Figure 6 (a) and (b), and we apply this semantic function, we will have respectively the product line M_F(fm1) and M_F(fm2).

M_F(fm1)= {{*f1,f2,f3,f4,f7*}, {*f1,f2,f3,f4,f7,f6*}, {*f1,f2,f3,f5,f7*}, {*f1,f2,f3,f5,f7,f6*}}

M_F(fm2)= { {*f1,f2,f4*}, {*f1,f2,f5*}, {*f1,f2,f4,f3,f6*}, {*f1,f2,f5,f3,f6*}, }

Figure 7 depicts an illustration of the OVM's semantic function proposed by Metzger et al. [6]. In this figure, each different OVM diagram that comply with the OVM's abstract syntax are assigned by the semantic function to each product line in the semantic domain. The semantic function is $\mathcal{M}_{OVM-M} : \mathcal{L}_{OVM-M} \rightarrow PL$, where $PL = \mathcal{PP}(VP \cup V)$. In this case, Metzger et al. define that in OVM a product line is defined like a combination of VPs and Vs. For example, if we have the two diagrams of the Figure 8 (a) and (b), and we apply the semantic function, we will have respectively the product line M_{OVM-M}(ovm1)

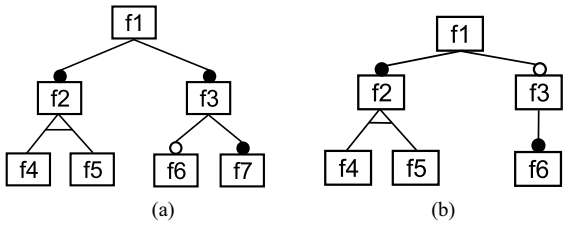


Figure 6: Concrete syntax of fm1 (a) and fm2 (b)

and M_{OVM-M}(ovm2).

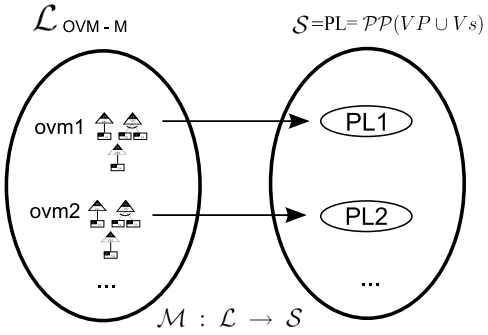


Figure 7: Semantic function of OVM-M

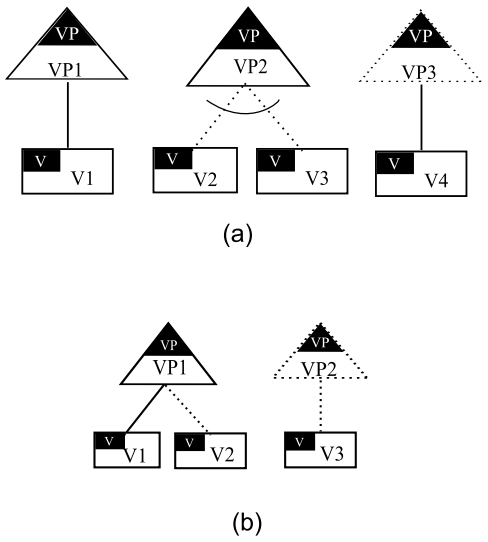


Figure 8: Concrete syntax of ovm1 (a) and ovm2 (b)

M_{OVM-M}(ovm1)= {{*VP1,V1,VP2,V2*}, {*VP1,V1,VP2,V2,VP3,V4*}, {*VP1,V1,VP2,V3*}, {*VP1,V1,VP2,V3,VP3,V4*}}

M_{OVM-M}(ovm2)= { {{*VP1,V1*}, {*VP1,V1,V2*}, {*VP1,V1,V2,VP2*}, {*VP1,V1,V2,VP2,V3*}, {*VP1,V1,VP2*}, {*VP1,V1,VP2,V3*}}

If we consider that a semantic domain of OVM is $\mathcal{PP}(V)$, we have another semantic function. But, as we already have the semantic function to the semantic domain $\mathcal{PP}(VP \cup V)$, we can achieve the semantic domain $\mathcal{PP}(V)$ excluding all VPs of the products. For example, the product line M_{OVM-M}(d2) would be

M_{OVM-M}(d2)= {{***V1***}, {***V1,V2***}, {*V1,V2,V3*}, {*V1,V3*}}

We can notice that with this semantic domain ($\mathcal{PP}(V)$) we have 4 products instead of 6, because two of them are duplicated {*V1*} and {*V1,V2*}. This happens because of the Optional VP2. When we consider that the VPs are part of the products, and in the model we have a optional VP with an optional child, we will have two products that are the same when implemented. For example, the products {*VP1,V1*} and {*VP1,V1,VP2*}. In fact the functionality that will be implemented will be V1, both products are the same.

To discuss about the difference between both OVM's semantic domain, we will use as an example the equivalence problem discussed in [8]. The equivalent models operation checks whether two models are equivalent. Two models are equivalent if they represent the same set of products [2]. According to OVM-M, if we observe the example depicted in the Figure 9, we can say that both models are equivalent, because they represent the same set of products. In the product of the OVM_1, *Media* is a *variation point* and in OVM_2, *Media* is a *variant*. In this example we have considered that the semantic domain of OVM was $\mathcal{PP}(VP \cup V)$, then the models seem to be equivalents because they represent the same set of products: {*Media, MP3, MP4*} = {MP3, Media, MP4}.

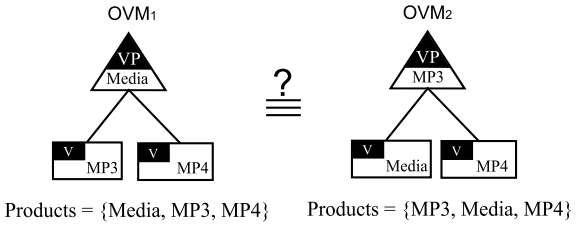


Figure 9: Equivalent models?

But, if we consider that the semantic domain of OVM is $\mathcal{PP}(V)$, then the models are not equivalents because they represent different set of products, {*MP3, MP4*} $\neq$ {Media, MP4}.

5. Conclusion and future work

The main contribution of this paper is to go forward in the discussion about the proposal existent in the literature regarding the formalization of OVM. We want to clarify what are the main differences between FMs and OVM to avoid future misinterpretation. There are differences between their abstract syntax like, the sort of nodes, the graph structure, types of information that capture, and the constraint that can be specified. On the other hand, in spite of

their semantic domain is considered the same, i.e a set of sets of combinations of nodes ($\mathcal{PP}(nodes)$), we consider that should be possible define the semantic domain of OVM as a set of sets of combinations of variants ($\mathcal{PP}(V)$). We think that we need to find out what is the most adequate semantic domain to deal with OVM in order to design a reasoning tool.

We trust that a well understood formal language is the starting point for our future work toward a safe automated reasoning tool for analysis of OVM models. In order to provide this tool, the next step of our work is to specify all the analysis operations that may be applied to OVM and formally define them.

6. Acknowledgement

We would like to thank David Benavides and Antonio Ruiz Cortés for their constructive comments. This work was partially supported by Spanish Government under CICYT project Web-Factories (TIN2006-00472) and by the Andalusian Government under project ISABEL (TIC-2533) and Evangelischer Entwicklungsdienst e.V. (EED).

References

[1] D. S. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference*, volume 3714 of *Lecture Notes in Computer Sciences*, pages 7–20. Springer–Verlag, 2005.

[2] D. Benavides. *On the automated analysis of software product lines using feature models*. PhD thesis, University of Sevilla, 2007.

[3] D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[4] P. Heymans, P.-Y. Schobbens, J.-C. Trigaux, Y. Bontemps, R. Matulevicius, and A. Classen. Evaluating formal properties of feature diagram languages. *IET Software*, 2(3):281–302, June 2008.

[5] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature–Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[6] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 243–253, 2007.

[7] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Fundations, Principles and Techniques*. Springer–Verlag, Berlin, DE, 2005.

[8] Roos-Frantz and S. Segura. Automated analysis of orthogonal variability models. a first. In *1st Workshop on Analyses of Software Product Lines (ASPL08)*, 2008.

[9] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, Feb 2007.

[10] M. Sinnema and S. Deelstra. Classifying variability modeling techniques. *Information & Software Technology*, 49(7):717–739, 2007.

Some Challenges of Feature-based Merging of Class Diagrams

Germain Saval
PReCISE Research Center
Faculty of Computer Science
FUNDP, University of Namur
gsa@info.fundp.ac.be

Jorge Pinna Puissant
Software Engineering Lab
University of Mons-Hainaut (U.M.H.)
Jorge.PinnaPuissant@umh.ac.be

Patrick Heymans
PReCISE Research Center
Faculty of Computer Science
FUNDP, University of Namur
phe@info.fundp.ac.be

Tom Mens
Software Engineering Lab
University of Mons-Hainaut (U.M.H.)
tom.mens@umh.ac.be

Abstract

In software product line engineering, feature models enable to automate the generation of product-specific models in conjunction with domain "base models" (e.g. UML models). Two approaches exist: pruning *of a large domain model, or* merging *of model fragments. In this paper, we investigate the impact of the merging approach on base models, and how they are made and used. We adopt an empirical method and test the approach on an example. The results show several challenges in the way model fragments are written, the need for new modelling language constructs and tool support.*

1. Introduction

A *Software Product Line* is *"a set of software-intensive systems that share a common, managed set of features satisfying 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"* [1]. *Software Product Line Engineering (SPLE)* is a rapidly emerging software engineering paradigm that institutionalises reuse throughout software development. By adopting SPLE, one expects to benefit from economies of scale and thereby lower the cost but also improve the productivity, time to market and quality of developing software.

Central to the SPLE paradigm is the modelling and management of *variability*, i.e., "*the commonalities and differences in the applications in terms of requirements, architecture, components, and test artefacts*" [2]. In order to tackle the complexity of variability management, a number of supporting modelling languages have been proposed.

An increasingly popular family of notations is the one of *Feature Diagrams* (FD) [3]. FDs are mostly used to model the variability of application "features" at a relatively high level of granularity. Their main purposes are (1) to capture feature commonalities and variabilities, (2) to represent dependencies between features, and (3) to determine combinations of features that are allowed or forbidden in the SPL.

Because FDs can be equipped with a formal semantics [4], they can be integrated into a model-driven engineering approach [5] and used to automatically generate (a set of) models specifying particular products from the product family, *the product models*. There are two basic approaches to generate product models:

1. a *pruning* approach where a global domain

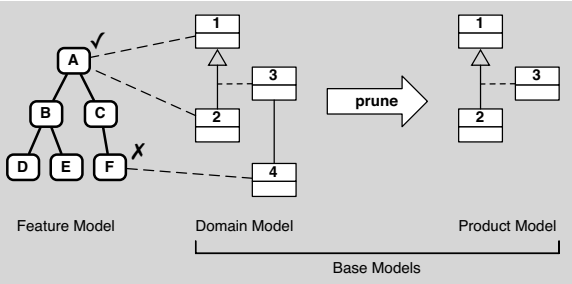


Figure 1. Pruning of a large model

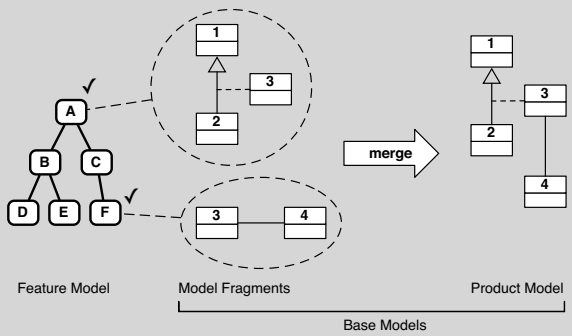


Figure 2. Merging of model fragments

model is tailored to a specific product by removing model elements from a feature model configuration (Figure 1);

2. a *merging* approach where different models or fragments, each specifying a feature, are combined to obtain a complete product model from a feature model configuration (Figure 2).

Our research question can be stated as follows: *when specifying static properties of features and generating a product model from a configured feature diagram, what are the challenges faced by the analyst using a merging approach?*

The rest of this paper is organised as follows. In Section 2.1, we will give an overview of the techniques proposed in the literature for model pruning, and in Section 2.2 for model merging. In Section 3, our example and the experimental settings will be presented. In the following sections, each identified challenge will be stated and discussed: The problem of synchronising different model fragments will be discussed in Section 4; the absence of variability notation in base models in Section 5; and the determination of the scope of a model fragment in Section 6. Requirements for better tool support will be suggested in Section 7. Section 8 will be devoted to a general discussion of our findings and future works will conclude this paper in Section 9.

2. Two generative approaches

2.1. Feature-based model pruning

Gottschalk *et al.* [6] favor a pruning approach to deal with dynamic aspects. They propose to configure domain models expressed by workflows (Petri nets). Their pruning algorithm comprises three steps: (1) removing elements that were not selected, (2) cleaning obsolete elements that are now disconnected, (3) check that every element is on a path from workflow input to output. Their approach is however not specific to SPL and does not use feature models.

Czarnecki *et al.* [7] also use a pruning approach. Each element of an activity diagram is annotated with a presence condition, expressed in terms of features. A FD is used to configure the activity diagram and a "flow analysis" ensures that each element is on a valid path and that the types of object flows are compatible. The same technique is used to configure the associated data model. Schätz [8] proposes a similar although less general approach based on reactive components that combine a domain-specific model (automata, component diagrams and application-specific conceptual model) and a variability model.

2.2. Feature-based model merging

Sabetzadeh *et al.* [9] use model merging to detect structural inconsistencies. They transform a static model into a graph and then into a relational model, *i.e.,* a textual description of the model. The consistency checks are expressed as a query on this relational model. Model merging is performed with the help of an interconnection diagram, which specifies semantic equivalence groups between model elements from different models. Traceability of model elements is kept along the

way, enabling to identify the origins of a detected inconsistency. In [10, 11, 12], the authors address dynamic models with behavioural matching as well. They provide algorithms and tool support to merge base models. Their work is not targeted on SPLE but, as we will see, is applicable here.

On the other hand, Perrouin *et al.* [13] specifically target SPLE. They propose to derive a product model by merging UML class diagram fragments. Their approach consists of two steps. First, given a feature model, a set of core assets and composition constraints, they merge model elements (e.g., classes) based on signature matching. The signature of a model element is defined as a set of syntactic properties for the element type, and can be reduced to its name. Second, the merged model can be customised to support additional features that were not included in the product family.

3. Testing Perrouin *et al.* merging approach

The experiment presented here followed the merging approach by Perrouin *et al.* [13]. The latter was chosen because it is integrated, model-driven and focused on SPLE. This experiment constitutes a first step towards comparison of the pruning and merging approaches, and further development and improvement of those. The chosen approach do not propose a specific merging algorithm and was complemented with the merging techniques of Sabetzadeh *et al.* [9].

3.1. The Conference Management System example

Through the rest of the paper we will use the example of a conference management system (ConfMS). A ConfMS is a software system that assists the Organising Committee of a scientific conference in the different phases of the conference organisation: publicise conference information like the Call for Papers, manage the submission and the review of the papers, organise the conference event locally, (*i.e.* the schedule, the sessions, the rooms), and publish the proceedings.

The IEEE [14] defines a conference as a *"major meeting which covers a specialised (vertical) or broad range (horizontal) set of topics (...) The program of a conference is designed to provide maximum opportunity for presentation of high quality papers appropriate to the defined scope of the conference. To this end, a Call for Papers is issued to attract the most qualified presenters possible. Presentations are accepted after appropriate peer review."*

The authors' knowledge of the ConfMS domain comes from another experiment meant to select and evaluate software [15], leading to the construction of several domain models. Figure 3 presents a feature diagram of such a ConfMS. The constructions used in this diagram are: features (rounded boxes), the usual *and*-decomposition (edges), optional nodes (hollow circles), *xor*-decomposition (edges with an arc), a *requires* constraint (thick arrow) and cardinalities (between curly braces). The features in white concern the review phase of conference organisation, we will specify them with a class diagram and obtain models for different products using the merging technique of Sabetzadeh *et al.* presented in section 2.2.

The *PC Organisation* feature represents the hierarchical layout of the programme committee (PC): the presence of a single PC or of multiple PCs (*Single PC* or *One PC per Track*) and the presence of a Review Board (RB) that oversees the work of the PC. The *Reviewing Process* feature describes how the different reviewing steps are laid out in sequence (*One Step* or *Two Steps*), if reviewers can delegate their reviews to others (*Delegation*) or if authors can comment the reviews (*Rebuttal*). The *Review Discussion* feature represents the possibility for reviewers to discuss the papers. The *Discussion Medium* feature represents the different means of discussion (by *Meeting* or via *Electronic* forum). The *Acceptance* feature represents the acceptance decision process for each paper. The list of accepted papers can be decided after discussion (*By Discussion*) by the PC (*Of PC*) or by the RB (*Of RB*), or by the Programme Chair alone (*By PCC*).

3.2. The experimental settings

The experiment was conducted by the two first authors, both PhD students who are knowledge-

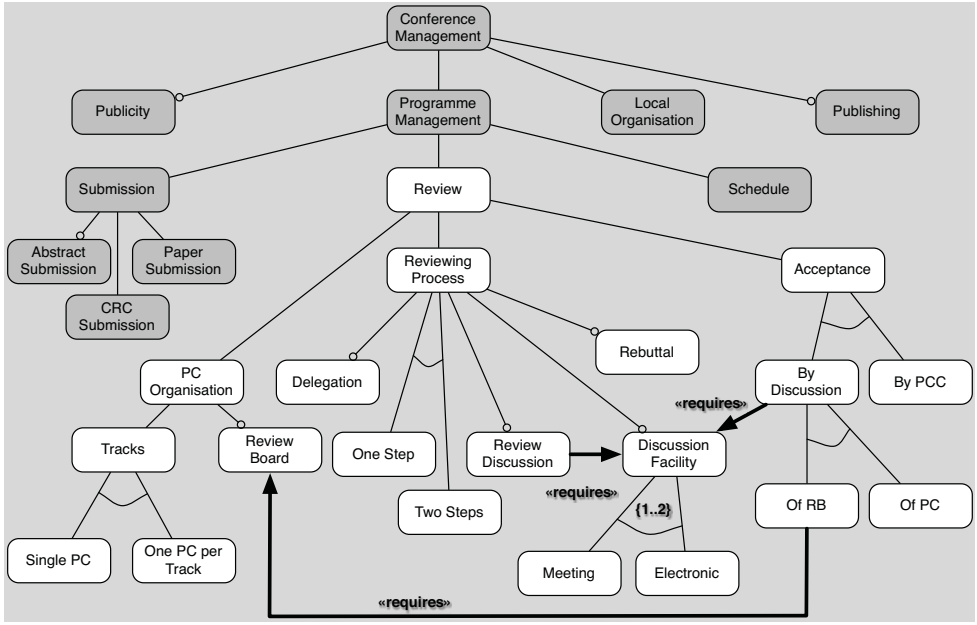


Figure 3. Conference Management System Feature Diagram

able in UML and feature modelling techniques, during ten eight-hour working days, using only an erasable white board, pens, generic diagramming tools (Poseidon for UML and OmniGraffle) and coffee.

The authors wrote the base class diagram presented in Figure 4, which models the commonalities of all the products of the feature diagram of Figure 3. They then wrote a class diagram fragment to model each sub-feature of the *Review* feature. The base diagram was completed iteratively by detecting the common model elements in every model fragment.

Although the general framework of Perrouin *et al.* [13] was followed, the merging algorithm itself used to generate these diagrams was executed manually and based on syntactic name matching inspired by Sabetzadeh *et al.* [9]. Equivalence groups between model elements are easier to determine in the experimental settings, instead of writing transformations inside Perrouin *et al.* [13] tool, and gives greater flexibility to test different solutions.

The first product generated by merging model fragments $P_1 = \{$*Review*; *PC Organisation*; *Tracks*; *Single PC*; *Reviewing Process*; *One Step*; *Acceptance*; *By PCC*$\}$ suits a small conference or a workshop, where there is a single PC and the acceptance decision is taken by the Programme Chair.

The second product $P_2 = \{$*Review*; *PC Organisation*; *Tracks*; *Single PC*; *Review Board*; *Reviewing Process*; *Delegation*; *One Step*; *Rebuttal*; *Review Discussion*; *Discussion Medium*; *Electronic*; *Meeting*; *Acceptance*; *By Discussion*; *Of RB*$\}$ suits a bigger conference where a Review Board supervises the reviewing of the PC and the decision is taken by this Review Board. The software should provide electronic and live meeting discussion facilities and allow review delegation.

Several challenges surfaced from this experiment, both during domain modelling and during the product model generation. In the next sections, we will detail three of them. Each is illustrated by the problems we faced during the experiment. Each of the following sections is subdivided as follows: firstly, the context in which a challenge appears is explained; secondly, we give specific instances encountered during the experiment, how we tried to overcome the problem and what are the alternatives available in the state of the art; finally,

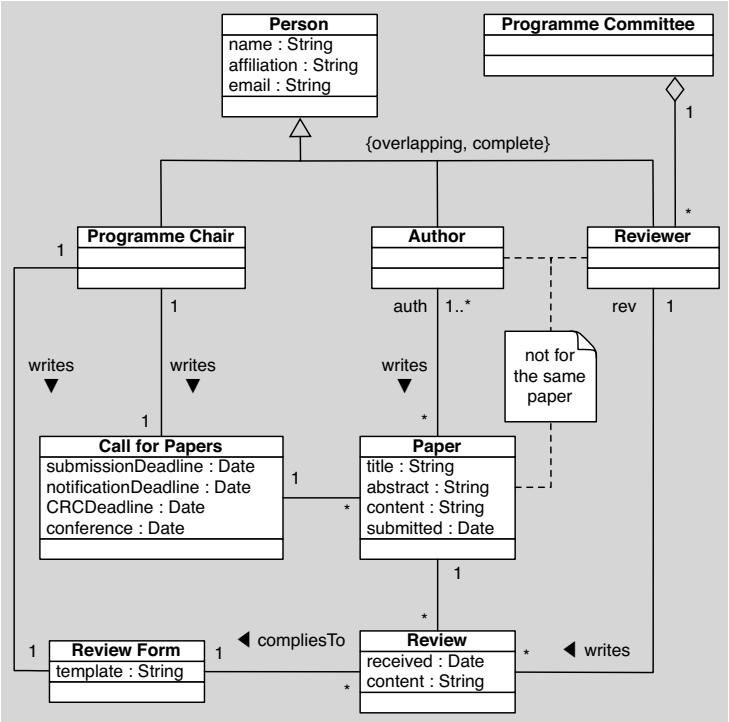


Figure 4. Base Class Diagram

we try to discuss the remaining issues and suggest improvements.

The order in which the challenges are presented was chosen only to facilitate the reader's comprehension and do not follow any order of importance or frequency. Those challenges were only selected among others because they had an important impact on the modelling process. Other challenges will be discussed in Section 8.

4. Challenge 1: distributed modelling and the need for synchronisation

4.1. Context: diverging base models

A first model comprising only the common concepts of the ConfMS was drawn. Then each feature was modelled successively. For a larger application however, it is likely that several features will be modelled in parallel. Remarkably, in both cases, the modelling process imposes some synchronisation to update the base models (it is a case of co-evolution of models). The use of a common terminology or, at least, a common understanding between the teams is therefore necessary. Especially since models are coupled and features interact with each other, it is important to achieve some level of agreement to be able to successfully merge the model fragments.

4.2. An instance

The two fragments (**F1** and **F2**) made of a set of interrelated classes shown in Figure 5 describe two different types of discussion. The *Review Discussion* feature (**F1**) offers reviewers the possibility to discuss the paper and their review. The *By Discussion of Review Board* feature (**F2**) offers to the *Review Board* the possibility to discuss the acceptance decision of a paper.

F1 and **F2** have a *common* part ($\mathbf{F1} \bigcap \mathbf{F2}$) and *different* parts $\mathbf{F1} \bigtriangleup \mathbf{F2} = (\mathbf{F1} - \mathbf{F2}) \bigcup (\mathbf{F2} - \mathbf{F1})$. After merging the fragments, the resulting class diagram contains the common parts ($\mathbf{F1} \bigcap \mathbf{F2}$) and the different parts ($\mathbf{F1} \bigtriangleup \mathbf{F2}$). The latter are asso-

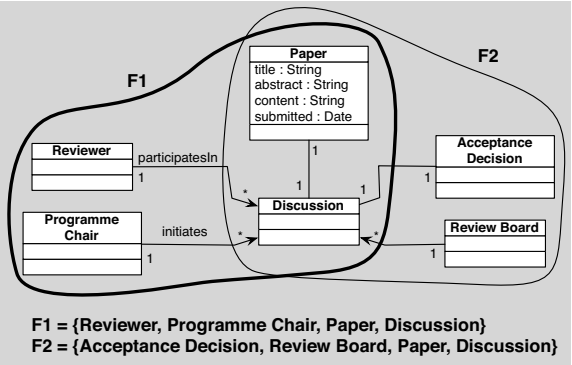


Figure 5. Merging of Two Features Class Diagram Fragment

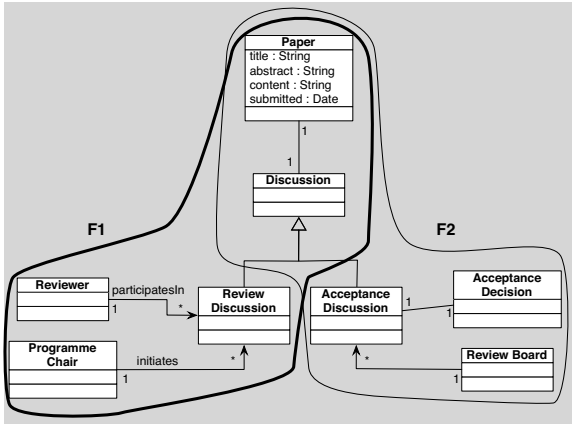


Figure 6. Merging of Two Features Class Diagram Fragment with class hierarchies

ciated to the common part. In this case, they are associated to the *Discussion* class.

The resulting class diagram is syntactically correct but it represents two very different situations (namely two different kinds of discussion) as if they were the same. In order to avoid this kind of inconsistency, a decision of the analyst is necessary. One solution (Figure 6) is to use class specialisation and create a sub-class for each type of discussion (Review Discussion and Acceptance Discussion) that is associated to each different part, and a super-class Discussion that is associated to the common parts.

4.3. Discussion

This is a modelling and a methodological problem. We followed an iterative process. That is, we pushed common elements in fragments associated to features higher in the feature tree when they were identified in several fragments. Conversely, we decided that common elements were shared down the feature tree following the feature decomposition relation in FDs. However, a single class can appear in several fragments. When it is concurrently modified, the status of the modifications is unclear. It can represent an undetected commonality or require a refactoring in several fragments if the concepts are actually different. For example, the *Discussion Facility* feature was identified early on as a common feature, but when the two different types of discussion were later modelled, this feature had to decomposed and the fragments associated to three features had to be modified to avoid confusion during the merging operation if the two discussion features were selected.

One of the proposed solutions is to use an integrated meta-model that blends feature models and base models. It allows to support feature-aware modelling and change propagation, because each model element can be annotated with the feature to which it pertains. Bachmann *et al.* [16] have suggested an integrated meta-model that can better support this approach. Such model can also simplify the merging algorithm, as Brunet *et al.* [12] noted. The general problem of detecting common concepts between static models is not new, however. It has been extensively studied in the case of database schema integration [17, 18]. It is also possible to detect this problem earlier by performing a partial merge of model fragments, preferably automatically, in a way similar to Sabetzadeh *et al.* [9].

5. Challenge 2: when variability notation is necessary in base diagrams

5.1. Context: variation points in base models

A model fragment can be incomplete before the feature model is configured because some model elements depend on specific configuration,

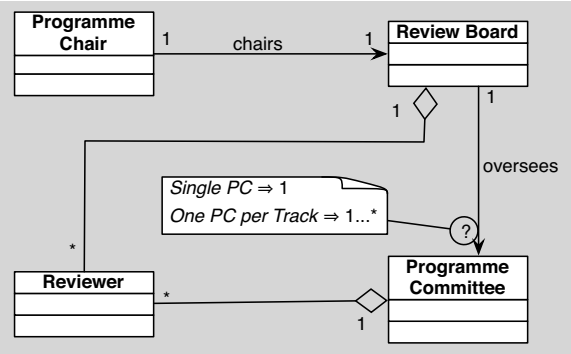


Figure 7. Review Board Class Diagram Fragment

i.e. the selection of certain features. Therefore, variability has to be explicitly modelled in base models, to be later resolved when the product model is generated by merging. More generally, some design decisions cannot be made *a priori* but the information is known when a specific product is built.

5.2. An instance

For example, the fragment associated with the *Review Board* feature is represented in Figure 7. We had to annotate it because a multiplicity was undefined. The multiplicity of the association *oversees* between the classes Review Board and Programme Committee can vary. This is because it depends on the selection of another feature: one of the two mutually exclusive decompositions of the *Tracks* feature.

5.3. Discussion

Some variability notation is necessary to indicate a decision point in the model, particularly when modelling an optional feature. UML is easily extensible and such information can be represented by UML comments. However, this solution seems to be impractical when the size of the product family increases. The major requirement is for this variability notation to be easily stored, retrieved and interpreted by software during modelling and merging. Several authors have identified this problem.

Pohl *et al.* [2] do not propose a general technique but use ad-hoc textual or graphical notations when necessary. Gomaa [19] uses UML stereotypes and parameters to annotate common elements and variability in diagrams. Those techniques are not specific to the approach studied here and are not formally defined to enable automation. Czarnecki *et al.* [7] propose an elegant solution: to attach to certain base model elements a formally defined presence condition expressed in terms of features (selected or not). This approach scatters product family variability information throughout the fragments and risks to defeat the purpose of a separate feature model, although this risk can be mitigated by a good visualisation tool.

6. Challenge 3: to what feature does a fragment belong?

6.1. Context: identification of atomic sets

When modelling a particular feature, the question of what is exactly modelled surfaces frequently. A specific feature with a well defined boundary within the system is easy, but other features are more cross-cutting by nature and the exact impact on the overall system is harder to define. In numerous occasions during the experiment, the authors wanted to be able to share a common model element between fragments, or modify a common element and specialise it. Other fragments were obviously associated to a set of features instead of a single one. Finally some features were more easily modelled in conjunction with others.

6.2. Instance

When a commonality is identified between features that represent a decomposition of a parent feature, the common elements were "pushed up" in the feature tree in the parent feature model fragment. An atomic set [20, 21] is a set of features that always appear together in a product. For example, in Figure 8 the atomic set composed of *Review*, *PC Organisation*, *Tracks*, *Reviewing Process* and *Acceptance* is highlighted. It represents the core of the ConfMS application, so that when a common model element belongs to one of its features, it is in fact added to the model fragment associated with

the whole atomic set.

Another notable group of features in Figure 8 is related to the *Discussion Facility* feature. As seen in Section 4, it is easier to model it in conjunction with the two features that require it. Although they do not form an atomic set, it is actually easier to include them in the scope of the model fragment associated with *Discussion Facility*.

6.3. Discussion

To alleviate this problem, and because the size of the domain model was moderate, we iteratively checked each completed fragment with the others, and tried to merge it to detect possible inconsistencies in advance. This solution, if not directly related to SPLE, was inspired by [9]. But the modelling of fragments also had an impact on the feature model: the discovery of possible ambiguity led to the modification of the FD and to reconsider the commonality of the product line, such as with the *Discussion* feature. These questions are mainly methodological and, although related to other domain modelling problems, specific to the merging approach. As far as we could observe, they are not yet covered in the literature. Concerning the merging algorithm, if model fragments are associated to sets of features instead of individual features, it will decrease the computational complexity for this, as well as for other automations (e.g. generation of all products or checking satisfiability).

7. Towards tool support

From the three challenges presented above, we can list several functionalities that would significantly improve the *modelling* of model fragments in a CASE tool supporting the approach: (1) an integrated meta-model encompassing feature model and base models; (2) the possibility to associate variability information in the form of presence conditions (boolean expressions on features) to every model element; (3) the identification of atomic sets and common features; (4) the possibility to associate model fragments to atomic sets and common features; (5) the sharing of common model elements in the relevant model fragments; (6) the specialisation of common elements into feature-specific fragments; (7) conversely, the factorisation (up in the feature tree) of common model elements identified along the modelling process; (8) an advanced visualisation engine that can selectively display the fragments associated to some features and the condition in which these fragments will appear in a product.

Some functionalities would also improve the merging operation: (1) a formally defined and machine-readable presence condition language; (2) traceability information between features and model elements; (3) a partial merge algorithm to detect common model elements or possible merging inconsistencies in advance.

8. General discussion

There are several threats to the validity of this study: the size of the example is moderate and some problems that would appear in bigger models may not be noticeable here; the experiment was performed manually (except for generic diagramming tools) due to the lack of a proper integrated tool supporting the approach. Although the researchers who carried out the experiment were trained in modelling with FD and class diagrams, this was the first time they used those languages in an integrated fashion. Hence, some challenges might have been emphasised by their lack of experience. However, such challenges would still be valuable to pinpoint because they highlight issues to be addressed when training new modellers to this integrated way of modelling. These challenges are likely to remain relevant for bigger products and families, due to the increased complexity of the modelling process execution (more products) and of the products themselves (more features).

The problems we have identified can be classified in three categories: (1) semantic, (2) methodological and (3) practical problems. The first category comes from the particular status of model fragments. They can express a limited amount of information, be incomplete, or even be syntactically incorrect and therefore, strictly speaking, meaningless but have an impact on the semantics of a product. The second category comes from

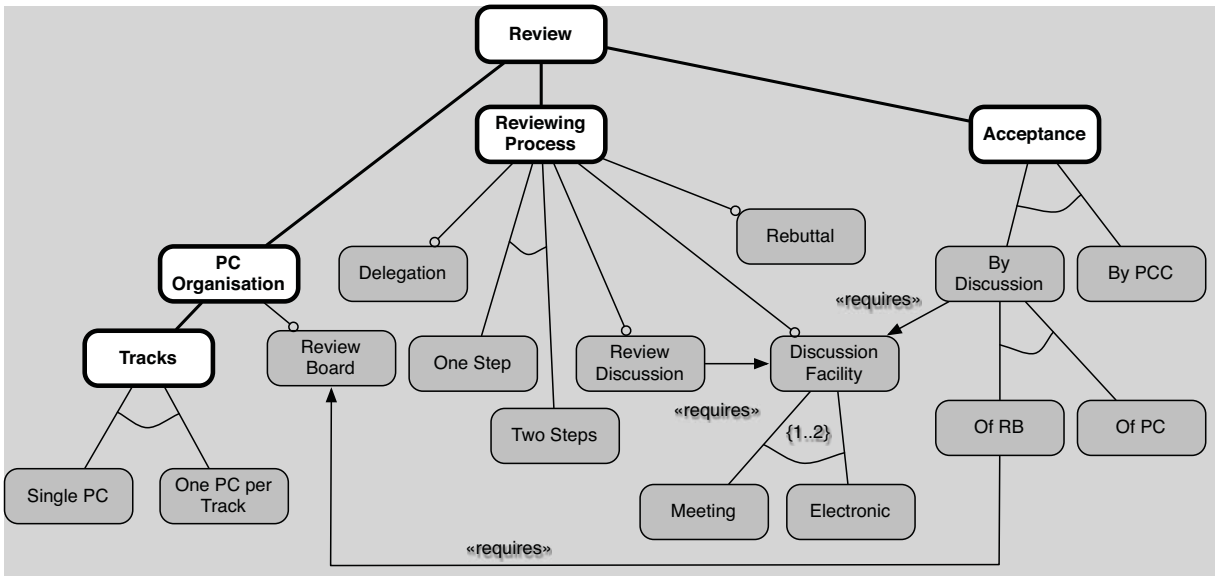


Figure 8. An atomic feature set in the Conference Management Set

the iterative and distributed nature of the process. Although feature modelling supports the separation of concerns, some synchronisation between the different model fragments is necessary from time to time, which requires to keep a view on the whole system and all of its variants, or locally on some set of features, which helps to inform particular design decisions. Finally, better tool support is necessary to ensure that the model fragments remain syntactically and semantically consistent with each other.

There are also advantages to such a merging approach. The ability to work on a subset of the features reduces the complexity of the problem, especially if it is highly decomposable, that is when features are interacting through a small and precisely defined interface. This approach can be partially supported by a tool. For static aspects, a simple name matching algorithm appears to cover most needs.

9. Conclusion & future works

We have reported three challenges that we faced during a modelling experiment. The first challenge was the lack of methodology to ease the co-evolution of model fragments, when common model elements are identified and factored, or a new understanding of the domain requires to specialise a common model element in different ways. The second challenge was the lack of variability notation in base models and the difficulty to separate the variability information from the domain model. The third challenge was difficulty to define the scope of a model fragment, that is to determine what set of features it describes. From this experiment, requirements for a better tool support were suggested.

In the future, we intend to compare the merging approach with the pruning approach. We also want to extend this experiment to the dynamic (behavioural) aspects of the base models. Finally, we hope to improve tool support by implementing the suggested functionalities and provide methodological guidelines.

Acknowledgements

The research reported here was partly funded by the MoVES Interuniversity Attraction Poles Programme, Belgium – Belgian Science Policy.

This work was also funded in part by the *Actions de Recherche Concertées - Ministère de la Communauté française - Direction générale de*

l'Enseignement non obligatoire et de la Recherche scientifique.

References

[1] P. C. Clements and L. Northrop, *A Framework for Software Product Line Practice - Version 4.2*. Pittsburgh, USA: Carnegie Mellon, Software Engineering Institute, 2003.

[2] K. Pohl, G. Bockle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, July 2005.

[3] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux, "Feature Diagrams: A Survey and a Formal Semantics," in *Proc. of the 14th IEEE International Requirements Engineering Conference RE'06*. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 136–145.

[4] J.-C. Trigaux, "Quality of feature diagram languages: Formal evaluation and comparison," Ph.D. dissertation, University of Namur, Faculty of Computer Science, September 2008.

[5] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[6] F. Gottschalk, W. M. van der Aalst, M. H. Jansen-Vullers, and M. La Rosa, "Configurable workflow models," *International Journal of Cooperative Information Systems (IJCIS)*, vol. 17, no. 2, pp. 177–221, June 2008. [Online]. Available: http://dx.doi.org/10.1142/S0218843008001798

[7] K. Czarnecki, "Mapping features to models: A template approach based on superimposed variants," in *GPCE'05, volume 3676 of LNCS*, 2005, pp. 422–437. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.88.6127

[8] B. Schätz, "Combining product lines and model-based development," *Electronic Notes in Theoretical Computer Science*, Jan 2007. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1571066107003933

[9] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik, "Consistency checking of conceptual models via model merging," in *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, 2007, pp. 221–230. [Online]. Available: http://dx.doi.org/10.1109/RE.2007.18

[10] M. Sabetzadeh, S. Nejati, S. Easterbrook, and M. Chechik, "A relationship-driven framework for model merging," in *MISE '07: Proceedings of the International Workshop on Modeling in Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, p. 2.

[11] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave, "Matching and merging of statecharts specifications," in *Proceedings of the 29th International Conference on Software Engineering, ICSE 2007*, ser. ICSE International Conference on Software Engineering, AT&T Laboratories-Research, Florham Park, NJ, United States, 2007, pp. 54–63. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.50

[12] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh, "A manifesto for model merging," in *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*. New York, NY, USA: ACM, 2006, pp. 5–12.

[13] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jézéquel, "Reconciling automation and flexibility in product derivation," in *Proceedings of the 12th International Software Product Line Conference SPLC '08*, 2008, pp. 339–348. [Online]. Available: http://dx.doi.org/10.1109/SPLC.2008.38

[14] Institute of Electrical and Electronics Engineers, "IEEE Conferences Organization Manual," last accessed July 2006. [Online]. Available: http://www.ieee.org/web/conferences/mom/all_manual.html

[15] G. Saval, P. Heymans, P.-Y. Schobbens, R. Matulevičius, and J.-C. Trigaux, "Experimenting with the Selection of an Off-The-Shelf Conference Management System," Poster presented at the 1st Intl. Workshop on Variability Modelling of Software-intensive Systems (VaMoS), January 2007.

[16] F. Bachmann, M. Goedicke, J. C. S. do Prado Leite, R. L. Nord, K. Pohl, B. Ramesh, and A. Vilbig, "A meta-model for representing variability in product family development." in *Software Product-Family Engineering, 5th Int'l Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers*, ser. LNCS, vol. 3014. Springer, 2003, pp. 66–80.

[17] C. Batini, M. Lenzerini, and S. B. Navathe, "A comparative analysis of methodologies for database schema integration," *ACM Comput. Surv.*, vol. 18, no. 4, pp. 323–364, 1986.

[18] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *The VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[19] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures (The Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, July 2004. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0201775956

[20] S. Segura, "Automated analysis of feature models using atomic sets," in *Proceedings of the First Workshop on Analyses of Software Product Lines ASPL*, 2008.

[21] W. Zhang, H. Zhao, and H. Mei, "A propositional logic-based method for verification of feature models," in *Formal Methods and Software Engineering*. Springer Berlin / Heidelberg, 2004, pp. 115–130. [Online]. Available: http://www.springerlink.com/content/fn47t2dwe26d3d3b

Benchmarking on the Automated Analyses of Feature Models: A Preliminary Roadmap *

Sergio Segura and Antonio Ruiz-Cortés
Department of Computer Languages and Systems
University of Seville
Av Reina Mercedes S/N, 41012 Sevilla, Spain
{sergiosegura, aruiz} AT us.es

Abstract

The automated analysis of Feature Models (FMs) is becoming a well-established discipline. New analysis operations, tools and techniques are rapidly proliferating in this context. However, the lack of standard mechanisms to evaluate and compare the performance of different solutions is starting to hinder the progress of this community. To address this situation, we propose the creation of a benchmark for the automated analyses of FMs. This benchmark would enable the objective and repeatable comparison of tools and techniques as well as promoting collaboration among the members of the discipline. Creating a benchmark requires a community to share a common view of the problem faced and come to agreement about a number of issues related to the design, distribution and usage of the benchmark. In this paper, we take a first step toward that direction. In particular, we first describe the main issues to be addressed for the successful development and maintenance of the benchmark. Then, we propose a preliminary research agenda setting milestones and clarifying the types of contributions expected from the community.

1. Motivation

The automated analysis of feature models consists on the computer–aided extraction of information from feature models. This extraction is performed by means of analysis operations. Typical operations of analysis allow finding out whether a feature model is void (i.e. it represents no products), whether it contains errors (e.g. feature that cannot be part of any products) or what is the number of products of the software product line represented by the model. A wide range of analysis operations and approaches to automate them have been reported [5, 7].

Recent workshops [6] and publications [8, 11, 19, 21, 25, 26] reflect an increasing concern to evaluate and compare the performance of different solutions in the context of automated analyses of feature models. However, the lack of standard problems to perform these empirical tests often difficult getting rigorous conclusions widely accepted by the community. Experiments in this context are mainly ad-hoc and not public and subsequently not repeatable by other researchers. Thus, performance conclusions are rarely rigorous and verifiable. As a result, these conclusions can barely be used to guide further research hindering the progress of the different solutions and, in general, of the whole discipline.

A *benchmark* is a test (a.k.a. test problem) or set of tests used to compare the performance of alternative tools or techniques [22]. Benchmarks have contributed to the progress of many disciplines along the years providing a level playing field for the objective and repeatable comparison of solutions. From a technical standpoint, the usage of benchmarks leads to a rigorous examination of performance results. From these results, the strengths and weaknesses of each proposal are highlighted helping researchers to improve their solutions and identify new research directions. From a social standpoint, benchmarks promote the collaboration and communication among different researchers. As a result, these become more aware of the work carried out by their colleagues and collaborations among researchers with similar interests emerge naturally.

Developing a benchmark for the automated analyses of feature models could contribute to the progress of the discipline, both at the technical and the social level. This was one of the main conclusions of the first workshop on Analysis of Software Product Lines (ASPL, [6]). There, a number of attendants agreed on the need for a benchmark (i.e. set

*This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project Web-Factories (TIN2006-00472) and the Andalusian Government project ISABEL (TIC-2533)

of standard feature models) to evaluate our solutions in rigorous and widely accepted way.

Creating a benchmark requires a community to share a common view of the problem faced and come to agreement about a number of issues related to the design, distribution and usage of the benchmark. In this paper, we take a first step toward that direction. In particular, we first describe the main issues to be addressed for the successful development and maintenance of the benchmark. Then, we propose a preliminary research agenda setting milestones and clarifying the types of contributions expected from the community.

The remainder of this paper is structured as follows. In Section 2 we detail the open issues to be addressed for the successful introduction of a benchmark in the community of automated analyses of feature models. Section 3 presents a preliminary research agenda for the development and maintenance of the benchmark. Finally, we summarize our conclusions in Section 4.

2. Open issues

We identify a number of open issues to be addressed for the successful introduction of a benchmark in the community of automated analyses of feature models. Next, we describe them.

2.1. Are we ready for it?

Sim *et al.* [22] draw attention to two preconditions that should exist in order to be success when introducing a benchmark into a research community.

The first precondition requires a minimum level of maturity in the discipline. As evidence that this minimum level has been reached, a sufficient number of different proposals to be evaluated using the benchmarks should be already available. This would provide some guarantee of the future interest of the community in the benchmark. This is a relevant condition since the effort needed to introduce a benchmark into a discipline is significant [17, 22, 24]. The community should have a strong commitment to participate actively on its development and maintenance, e.g. proposing new test problems regularly.

The second precondition point out the need for an active collaboration among researchers. These should be well-disposed to work together to solve common problems. According to Sim, these collaborations help researchers to gain familiarity and experience creating a community more receptive to the results and consequently more likely to use the benchmark. Some evidences of the willingness to collaborate may be deduced from previous collaboration between the members of the community, e.g. multi-author publications.

A number of evidences suggest that these two conditions already exist in the domain of automated analyses of feature models. In the context of maturity, existing surveys [5, 7] reflect that a sufficient number of proposals to be evaluated using the benchmark are already available. Additionally, an increasing concern to evaluate and compare the performance of tools and techniques is detected in recent publications [8, 11, 19, 21, 25, 26]. In the context of collaboration, recent workshops [6] and multi-authors publications such as [4] or [25] (best paper award at SPLC'08) also suggest that the community is ready to incur in the development of a benchmark. Despite this, we consider that the introduction of a benchmark must be still further debated by the community in order to find out the level of interest and commitment of its members to participate on it.

2.2. Agreeing a format

Reaching a consensus on a language to specify test problems is a key point for a benchmark being accepted by the community. To this end, the semantic, abstract and concrete syntax of the language should be carefully studied. The semantic should be well defined to avoid ambiguity and redundancies in the specification of the problems. The abstract syntax should be flexible enough to enable the usage of the benchmark with tools and techniques using different notations. Finally, the concrete syntax should be as simple as possible to simplify its understanding and manipulation.

For the semantic and abstract syntax of the language, an overview of the available papers surveying feature modelling notations would be desirable. A good starting point could be the work of Schobbens *et al.* [20]. In their work, the authors survey a number of feature diagram notations and study some of their formal properties. As a result, they propose a new feature diagram language, VFDs (Varied Feature Diagrams), embedding all other variants.

For the concrete syntax, that is, the specific format used to represent and distribute the problems, we foresee two main options: plain text and XML. These appear to be the most popular input formats used in the existing feature model analyses tools. An example of tool using plain text is the Ahead Tool Suite[1] in which feature models are represented as grammars. Some examples of tools using XML are the Feature Model Plug-in[2], the FAMA framework[3] and Pure::Variants[4]. For the selection of one format or another, advantages and drawbacks of each option should be evaluated and debated. On the one hand, plain text formats tend to be shorter than XML documents and usually more suitable to be written by human beings. On the other hand,

[1] http://www.cs.utexas.edu/users/schwartz/ATS.html
[2] http://gp.uwaterloo.ca/fmp/
[3] http://www.isa.us.es/fama/
[4] http://www.pure-systems.com/

XML is a widely extended mechanism to exchange information easy to be defined (e.g. XML schema) and parsed.

A review of the formats used in related software benchmarks could also be helpful to support a decision. In a first outlook to these benchmarks we noticed that plain text seems to be the preferred format especially on those benchmarks related to mathematical problems. Some examples are the DIMACS CNF format [1] for satisfiability problems, the MPS format [2] for linear programming or the AMPLE format [10] for linear and nonlinear optimization problems. We also found some related benchmark dealing with XML format such as GXL [12], introduced in the context of reverse engineering, or XCSP [3] for constraint programming.

Finally, feature modelling community could also benefit from the lessons learned in other domains when selecting a format. We found in XCSP an interesting case of this. In the current version of the format (i.e. 2.1), released in January 2008 for the Third International CSP Solver Competition[5], the authors felt the need to distinguish between two variants of the format: a *'fully-tagged'* representation and a *'abridged'* one. According to the authors, the tagged notation is *'suitable for using generic XML tools but is more verbose and more tedious to write for a human being'* meanwhile the abridged notation *'is easier to read and write for a human being, but less suitable for generic XML tools'*. As a negative consequence of this, authors of XCSP must now provide up-to-date support for two different formats which include updating documentation, parsers, tools to convert from one representation to another, etc. Studying the synergies between XCSP and our future benchmark format could help us to predict whether we could find the same problem using XML. Reporting similar lessons learned in other domains would be highly desirable for supporting a decision.

2.3. Selection of test problems

The design of test problems is recognized as one of the most difficult and controversial steps during the development of a benchmark [22, 24]. Walter Tichy advises:

> *'The most subjective and therefore weakest part of a benchmark test is the benchmark's composition. Everything else, if properly documented, can be checked by the skeptic. Hence, benchmark composition is always hotly debated.'* [24] (page 36)

These test problems should be representative of the real problems to be solved by the tool or technique under test. As discussed in [9, 14], there are essentially three sources of test problems: those which arise in real scenarios, those that are specifically developed to exercise a particular aspect of the tool or technique under test and randomly generated ones. There is not a consensus about the criteria for the selection of one type or another [14]. In practice, researchers from different research disciplines usually adopt a pattern of use. There exist well-documented deficiencies of each alternative. In the case of specific collection of problems, Jackson *et al.* [14] summarizes them as follows:

- The test set is usually small compared whit the total set of potential test problems.
- The problems are commonly small and regular. There may exist large-scale problems but they are often not distributed because it is difficult and time-consuming.
- Problems may have similar properties. As a result of this, some of the features of the tool or technique could be not exercised.
- Optimizing a technique or tool for a set of test problems may not provide ideal performance in other settings.

Randomly generated problems overcome some of the drawbacks detailed previously but also attract other negative opinions. In particular, these critics focus on the lack of realism of those problems and the systematic structures that sometimes may appear on them.

Two main types of problems are reported in the context of feature models: invented and randomly generated ones. One the one hand, invented feature models are usually small and regular. They are used for research purposes but they rarely can be used to showcase the performance of a tool or technique. On the other hand, randomly generated ones are more adequate to check the performance of tools but they do not represent real problems and rarely can be replicated by other researchers. There exist references in the literature to software product lines with thousand of features [23] (page 32) but to the best of our knowledge associated feature models are not available. We presume this may due to the effort required to distribute them or to confidentiality issues.

Different types of contribution would be welcome by the community of automated analyses of feature models in the context of a benchmark. Firstly, feature models from real scenarios are highly desirable to both studying their properties and using them as motivating inputs for the tools and techniques under evaluation. Notice that these feature models could include not only feature models from industry but also feature models extracted from OS projects (e.g. [13, 16]). Secondly, collection of problems published in the literature would be helpful since they represent widely accepted problems by the community. Finally, random feature models will be needed to evaluate the performance of tools and techniques dealing with large-scale problems. Regardless the type of problem proposed, this should be clearly

[5] http://cpai.ucc.ie/

justified by stating what characteristics make it a good test problem and what it is hoped to learn as a result of running it.

2.4. Benchmark development

The key principle underlying the benchmark development is that it must be a community effort [22, 24]. Members of the discipline should participate actively in the development and maintenance of the benchmark through a number of tasks. Some of these are:

- Agreeing a format for the test problems.
- Design and publication of test problems.
- Usage of the benchmark and publication of results.
- Regular submission of new test problems.
- Report errors or possible improvements in the format or existing test problems.

Note that continued evolution of the benchmark is required to prevent users from optimizing their tools or techniques for a specific set of test problems.

Based on their experience, Sim *et al.* [22] attributes the success of a benchmark development process to three factors, namely:

- *'The effort must be lead by a small number of champions'*. This small group of people should be responsible of keeping the project alive and will be commonly in charge of organizing and coordinating activities to promote discussion among the members of the community.
- *'Design decisions for the benchmark need to be supported by laboratory work'*. Some experiments may be needed to show the effectiveness of a solution and to support decisions.
- *'The benchmark must be developed by consensus'*. To this end, it is necessary to promote the discussion of the members of the community in many formats as possible. Some options are workshops, conferences, mailing lists, discussion forums, Request for Comments (RFC), etc. In this context, Sim point at face-to-face meeting in conferences and workshops as the most effective method.

For the successful development of a benchmark, community should be aware of how they can contribute. To this end, detailed information about the different tasks to be carried out and the effort required for each of them would be highly desirable. This paper pretend to be a first contribution in that direction (see Section 3).

2.5. Support infrastructure

Successful benchmarks are commonly provided together with a set of tools and mechanism to support its usage and improvement. Some examples are mailing list to enable discussion among users, test problems generators, parsers, documentation, etc. These contributions are welcome at any time but they are especially appealing during the release of the benchmark in order to promote its usage within the community. Participating at this level may required an important effort from the community but it also may appear as good opportunity to come in contact with other researchers working in similar topics.

Contributions from the community of feature models in any of these forms (i.e. generators, parsers, etc.) will be greatly welcome.

2.6. Using the benchmark

Performing experiments and reporting its results is not a trivial task. The analysis, presentation and interpretation of these results should be rigorous in order to be widely accepted by the community. To assist in this process, a number of guidelines for reporting empirical results are available in the literature. Some good examples can be found in the areas of mathematical software [9, 14] and software engineering [15, 18].

At the analysis level, aspects such as the statistic mechanisms used, the treatment of outliers or the application of quality control procedures to verify the results should be carefully studied.

A number of considerations should also be taken into account when presenting results. As an example, Kitchenham *et al.* [18] suggest a number of general recommendations in the context of experiments in software engineering. These include providing appropriate descriptive statistics (e.g. present numerator and denominator for percentages) or making a good usage of graphics (e.g. avoid using pie charts).

Finally, the interpretation of results should also follow some well-defined criteria and address different aspects. These may include describing inferences drawn from the data to more general conditions and limitations of the study.

Contributions for the correct usage of the benchmark in the context of automated analyses of feature models would be helpful. These may include guidelines and recommendations about how to get (e.g. useful measures), analyse (e.g. adequate statistics packages), present (e.g. suitable graphs) and interpret (e.g. predictive models) the benchmark results.

3. Preliminary roadmap

Based on the open issues introduced in previous sections, we propose a preliminary roadmap for the development of a benchmark for the automated analyses of feature models. In particular, we first clarify the types of contributions expected from the community. Then, we propose a research agenda.

3.1. Types of contributors

The main goal of this section is to clarify the ways in which the community can contribute to the development and usage of the benchmark. To this end, we propose dividing up the members of the discipline interested in the benchmark into three groups according to their level of involvement in it, namely:

- **Users.** This group will be composed of the members of the discipline interested exclusively in the usage of the benchmark and the publication of performance results. Exceptionally, they will also inform about bugs in the test problems and tools related to the benchmark.
- **Developers.** This group will be composed of members of the community interested in collaborating in the development of the benchmark. In addition to the tasks expected from users, the contributions from this group include:
  - Designing and maintaining new test problems
  - Proposing guidelines and recommendations for an appropriate usage of the benchmark.
  - Developing tools and/or documentation, e.g. test problem generators.
- **Administrators.** These will be the *'champions'* in charge of most part of the work. This group will be composed of a few researchers from one or more laboratories. In addition to the tasks associated to the users and developers, the contributions expected from this group include:
  - Organizing and coordinating activities to promote discussion (e.g. performance competitions).
  - Proposing a format to be accepted by the community.
  - Publication of test problems.
  - Setting mechanisms to promote contributions from the community, e.g. template for submitting new test problems

3.2. Research agenda

We identify a number of tasks to be carried out for the development and maintenance of a successful benchmark for the automated analyses of feature models. Figure 1 depicts a simplified process model illustrating these tasks using BPMN[6] notation. Rectangles depict tasks (T-X) and diamond shapes represent decisions (D-X). For the sake of simplicity, we distinguish tree group of tasks: those carried out by the community (i.e. administrators, developers and users of the benchmark), those performed by the administrators and those tasks accomplished by both administrators and developers.

As a preliminary step, community of automated analyses of feature models should evaluate whether we are ready to incur in the development of a benchmark (*T-01*). As discussed in Section 2.1, we consider this discipline is mature enough and has the necessary culture of collaboration to start working on it. However, we still consider that a noticeable interest from the members of the discipline to participate either in the development or the usage of the benchmark should be detected. If this precondition is not met (*D-01*), it does not mean the benchmark cannot be used. Rather, it means that some actions should be then carried out to establish this precondition. In this context, Sim suggests waiting for more research results and planning activities to promote collaboration among researchers (*T-02*).

Once the community agrees on the need for a benchmark, the design of a format for the test problems should be the first step (*T-03*). This should be proposed by the administrators and approved by a substantial part of the community (*D-02*). It should be presented in a standard format such a technical report and include a versioning system to keep record of its evolution. Note that several attempts (i.e. versions) could be needed until reaching a wide consensus.

Once an accepted format is available, an initial set of test problems (*T-04*) and support tools (*T-05*) should be released by administrators. At the very least, we consider these should include a parser for the test problems and platform to publish the material related to the benchmark (e.g. FTP site). At this point, the benchmark would already be fully usable.

Finally, a number of contributions from the community for the maintenance and improvement of the benchmark would be expected. These include i) Using the benchmark and publishing the results (*T-06*), ii) Reporting bug and suggestions (*T-07*), iii) Organizing activities to promote discussion among the members of the community (*T-08*), iv) Proposing new test problems (*T-09*), v) Developing tools and documentation (*T-10*), and vi) Providing guidelines and recommendations for an appropriate usage of the benchmark (*T-11*).

[6] http://www.bpmn.org/

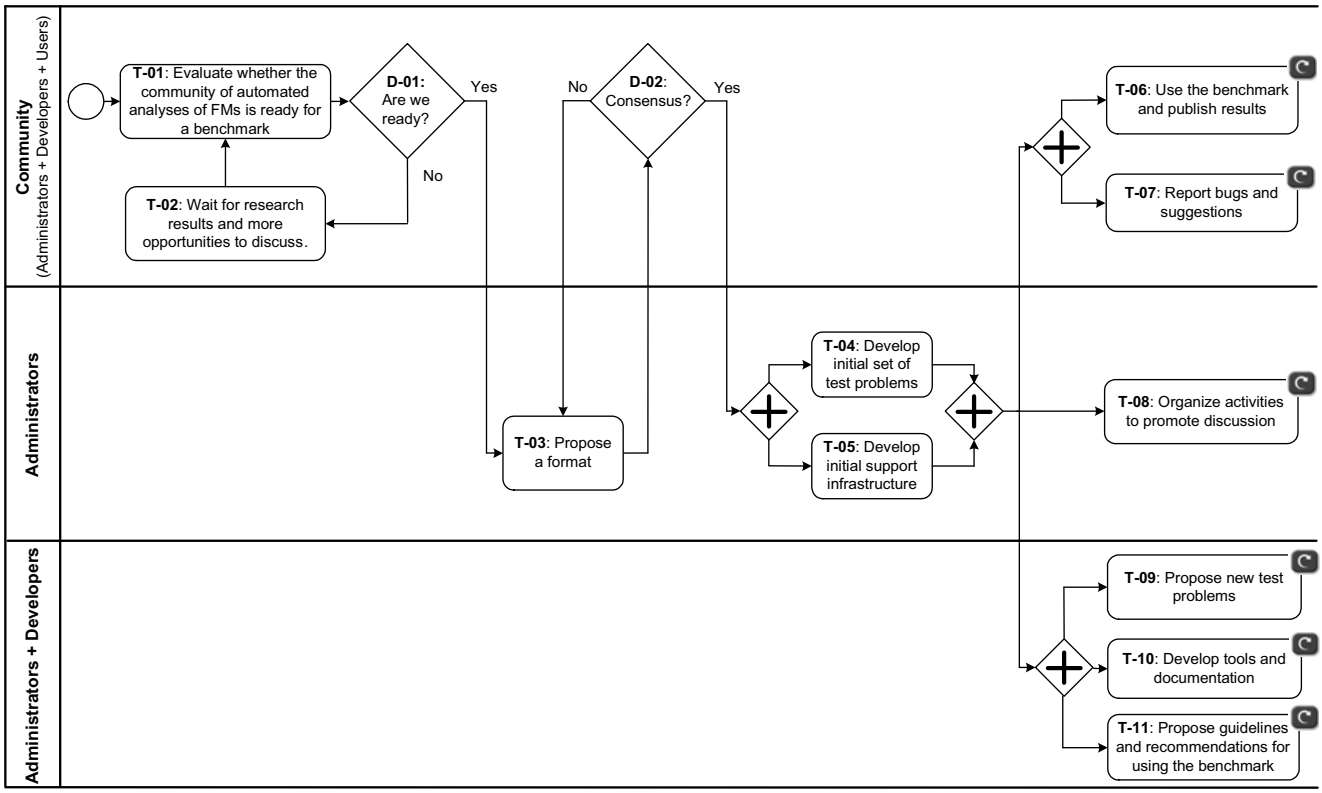


Figure 1. A process model of the proposed research agenda

4. Conclusions

The introduction of a benchmark for the automated analyses of feature models could contribute to the progress of the discipline by providing a set of standard mechanisms for the objective and repeatable comparison of solutions. A key principle underlying the creation a benchmark is that it must be a community effort developed by consensus. To this end, members of the discipline should first share a common view of the problem faced and the tasks to be carried out for its development. This is the main contribution of this paper. In particular, we first described the open issues to be addressed for the successful introduction of a benchmark for the automated analyses of feature models. Then, we proposed a preliminary roadmap clarifying the types of contributions expected from the community and the main steps to be taken. To the best of knowledge, this is the first contribution in the context of benchmarking on the automated analyses of feature models.

Acknowledgments

We would like to thank Dr. David Benavides whose useful comments and suggestions helped us to improve the paper substantially.

References

[1] DIMACS Conjunctive Normal Form format (CNF format). Online at http://www.satlib.org/Benchmarks/SAT/satformat.ps.

[2] Mathematical Programming System (MPS format). Online at http://lpsolve.sourceforge.net/5.5/mps-format.htm.

[3] XML Representation of Constraint Networks Format XCSP 2.1. Online at http://www.cril.univ-artois.fr/CPAI08/XCSP2_1.pdf.

[4] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, December:45–47, 2006.

[5] D. Benavides. *On the Automated Analysis of Software Product Lines using Feature Models. A Framework for Developing Automated Tool Support.* PhD thesis, University of Seville, 2007.

[6] D. Benavides, A. Ruiz-Cortés, D. Batory, and P. Heymans. First International Workshop on Analyses of Software Product Lines (ASPL'08), September 2008. Limerick, Ireland.

[7] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feture models. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2006.

[8] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, 2006.

[9] H. Crowder, R.S. Dembo, and J.M. Mulvey. On reporting computational experiments with mathematical software. *ACM Transactions on Mathematical Software*, 5(2):193–203, 1979.

[10] R. Fourer, D.M. Gay, and B.W. Kernighan. A modeling language for mathematical programming. *Management Science*, 36(5):519–554, 1990.

[11] A. Hemakumar. Finding contradictions in feature models. In *First Workshop on Analyses of Software Product Lines (ASPL 2008). SPLC'08*, Limerick, Ireland, September 2008.

[12] R.C. Holt, A. Winter, and A. Schürr. GXL: Toward a Standard Exchange Format. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 162, Washington, DC, USA, 2000. IEEE Computer Society.

[13] A. Hubaux, P. Heymans, and D. Benavides. Variability modelling challenges from the trenches of an open source product line re-engineering project. In *Proceedings of the Sofware Product Line Conference*, pages 55–64, 2008.

[14] R.H. Jackson, P.T. Boggs, S.G. Nash, and S. Powell. Guidelines for reporting results of computational experiments. report of the ad hoc committee. *Mathematical Programming*, 49(1):413–425, November 1990.

[15] A. Jedlitschka and D. Pfahl. Reporting guidelines for controlled experiments in software engineering. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 10 pp.+, 2005.

[16] C. Kastner, S. Apel, and D. Batory. A case study implementing features using aspectj. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 223–232, Washington, DC, USA, 2007. IEEE Computer Society.

[17] B.A. Kitchenham. Evaluating software engineering methods and tool. Part 1 to 12. *ACM SIGSOFT Software Engineering Notes*, 21-23(1), 1996-1998.

[18] B.A Kitchenham, S.L. Pfleeger, L.M. Pickard, P.W. Jones, D.C. Hoaglin, K.E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transaction on Software Engineering*, 28(8):721–734, August 2002.

[19] M. Mendonca, A. Wasowski, K. Czarnecki, and D. Cowan. Efficient compilation techniques for large scale feature models. In *GPCE '08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 13–22, New York, NY, USA, 2008. ACM.

[20] P. Schobbens, J.C. Trigaux P. Heymans, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, Feb 2006.

[21] S. Segura. Automated analysis of feature models using atomic sets. In *First Workshop on Analyses of Software Product Lines (ASPL 2008). SPLC'08*, pages 201–207, Limerick, Ireland, September 2008.

[22] S.E. Sim, S. Easterbrook, and R.C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 74–83, Washington, DC, USA, 2003. IEEE Computer Society.

[23] V. Sugumaran, S. Park, and K. Kang. Software product line engineering. *Commun. ACM*, 49(12):28–32, 2006.

[24] W.F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998.

[25] J. White, D. Schmidt, D. Benavides P. Trinidad, and Ruiz-Cortés. Automated diagnosis of product-line configuration errors in feature models. In *Proceedings of the 12th Sofware Product Line Conference (SPLC'08)*, Limerick, Ireland, September 2008.

[26] J. White and D.C. Schmidt. Filtered cartesian flattening: An approximation technique for optimally selecting features while adhering to resource constraints. In *First International Workshop on Analyses of Software Product Lines (at SPLC'08)*, Limerick, Ireland, September 2008.

Abductive Reasoning and Automated Analysis of Feature Models: How are they connected?*

Pablo Trinidad, Antonio Ruiz–Cortés
Dpto. Lenguajes y Sistemas Informáticos
University of Seville
{ptrinidad,aruiz} at us.es

Abstract

In the automated analysis of feature models (AAFM), many operations have been defined to extract relevant information to be used on decision making. Most of the proposals rely on logics to give solution to different operations. This extraction of knowledge using logics is known as deductive reasoning. One of the most useful operations are explanations that provide the reasons why some other operations find no solution. However, explanations do not use deductive but abductive reasoning, a kind of reasoning that allows to obtain conjectures why things happen. As a first contribution we differentiate between deductive and abductive reasoning and show how this difference affect to AAFM. Secondly, we broaden the concept of explanations relying on abductive reasoning, applying them even when we obtain a positive response from other operations. Lastly, we propose a catalog of operations that use abduction to provide useful information.

1. Introduction

The *automated analysis of feature models* (AAFM) intends to extract relevant information from *feature models* (FM) to assist on decision making and even to produce design models or code. The general process that most of the works propose to deal with automated analysis is transforming a FM into a logic paradigm and solving declaratively the problem. We have noticed that most of the proposed operations use deductive reasoning techniques to extract such an information. The way deductive reasoning works is obtaining objective conclusions from its knowledge base (KB) making explicit an implicit information.

*This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project Web-Factories (TIN2006-00472) and by the Andalusian Government under project ISABEL (TIC-2533).

But in some situations, it may be interesting not only obtaining conclusions but knowing the reasons why that conclusion is inferred. For example, if we find an error in a FM such as a dead feature we must be interested in the relationships that make this error appearing. So we can use this information to assist on error repairing. In case we are searching for the cheapest product to be produced in a family and we obtain a specific product, we may be searching for the relationships and criteria that have been taken into account. This transverse operation is commonly known in FM analysis community as *explanation* and may be used in conjunction with any deductive operation.

These two examples, remarks the automated analysis as a two-step process, where an information is extracted from a FM firstly by means of deductive reasoning, and just in case we are interested in obtaining further information, we may ask for the reasons why we have obtained such an information using abductive reasoning. As a first contribution of this paper we remark this difference, distinguishing between two kinds of operations: *deductive operations*, that use deductive reasoning to reach for a result; and explanatory or *abductive operations*, which use abductive reasoning to explain a result obtained from a deductive operation (see Figure 1). As a consequence, we have observed that most of the proposed operations in automated analysis are deductive operations, and abductive operations have only been proposed to solve particular problems such as obtaining explanations for void FMs and dead features. Therefore, and as a second contribution, we propose a catalog of abductive operations that broadens their field of action to be applied to the results of any deductive operation.

One of the main contributions in [2] is proposing a general transformation from a FM into many logic paradigm or solver such as Constraint Satisfaction Problems (CSP), satisfiability problems or Binary Decision Diagrams(BDD) by means of a formal description of the problem in the so called FAMA Theoretical Framework. However, his proposal was centered in deductive reasoning and explanations were proposed as an operation that did not fit into his de-

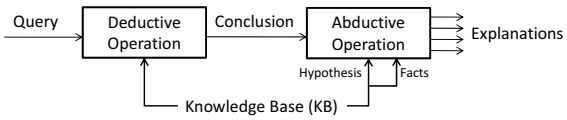


Figure 1. The Link between Deductive and Abductive Reasoning

ductive framework so solving them was considered to be an open issue. Now we know that explanations may never fit into his deductive framework as it is an abductive operation. However, we envision that we may follow the same structure than Benavides' FAMA Theoretical Framework for abductive operations so the problem may be represented in a theoretical level so several solvers and logic paradigms may be used to solve them. Therefore, as a last contribution, we envision how some of the current proposals in abductive reasoning may fit into such a framework and which are the solvers, techniques or algorithms that can be used to deal with abductive operations.

This paper is structured as follows: In Section 2 we briefly present a study of the works in the automated analysis of FM from the point of view of deductive and abductive reasoning. In Section 3, we introduce the concept of abductive reasoning more in depth, pointing out its relationship with diagnosis problems. The catalog of abductive operations is presented in Section 4. We envision the future works and research lines, exposing some conclusions in Section 5.

2 Background

2.1 Analysis of Feature Models

The automated analysis of FMs intends to extract relevant information from FMs to assist decision making during SPL development. To obtain such an information, many authors have proposed different operations for products counting, filtering, searching and error detecting that are summarized in a survey in [4]. Most of the proposals rely on declarative techniques and logics to extract information such as Constraint Satisfaction Problems (CSP) [3], SAT solvers [7] and Binary Decision Diagrams (BDD)[5].

In the works where logics are used to give a response to those operations, they use a common way of reasoning called deduction. Informally speaking, Deduction makes explicit an implicit information in a theory. It means that the only information that may be extracted from a model is the one that is modeled, and what we are doing when reasoning deductively about a FM is making explicit a hard-to-see information. For example, if we select feature A in the FM in Figure 2, deductive reasoning may reach the conclusion that feature C may not be selected. If we select features A and C deduction is only able to determine that there is no possible configuration containing both features at the same time. If we want to explain the reason why A and C are mutually exclusive, deductive reasoning is not the right choice.

2.2 Explanations in Feature Models

The need of explanations were firstly detected by Kang *et al.*[6] to determine the reasons why a FM is void. In this work, Prolog was proposed to model and explain void FMs if it were the case. Batory proposed in [1] using Logic Truth Maintenance Systems (LTMS) to explain why a configuration is not valid. Sun *et al.* [10] use Alloy Analyzer, a model checking tool based on first-order logic, to detect the sources of void FMs. Wang *et al.*[14] propose using description logic and RACER tool to deal with dead features and void FMs. Trinidad *et al.* describe in [13, 11] the errors explanation problem in terms of theory of diagnosis[9], dealing with different kinds of error. They propose a framework where different implementations were accepted and gave details about using constraint satisfaction optimization problems (CSOP) to deal with them. White *et al.*[15] proposed using CSOP to deal with invalid configurations.

Notice that the techniques proposed to search for explanations are different from those proposed to deal with deductive reasoning. Moreover, most of the proposals that deal with explanations focus on error analysis. We already presented in [11] a framework to deal with errors relying in diagnostic reasoning which is a particular application of abductive reasoning as we will remark in next Section.

2.3 Catalog of Deductive Operations

There are two main works [4, 2] that have summarized the state of the art in the automated analysis of FMs. Both of them present an exhaustive survey of the operations that have been proposed in the most relevant works.

- Determining if a product, feature model or configuration is valid.
- Counting and obtaining all the products.
- Calculating a feature commonality and variability and determining the core and variant features.
- Filtering and searching for optimal products.
- Dead and false-optional features and wrong-cardinalities detection.
- Explanations and error correction.
- Model transformations such as simplification and merging.

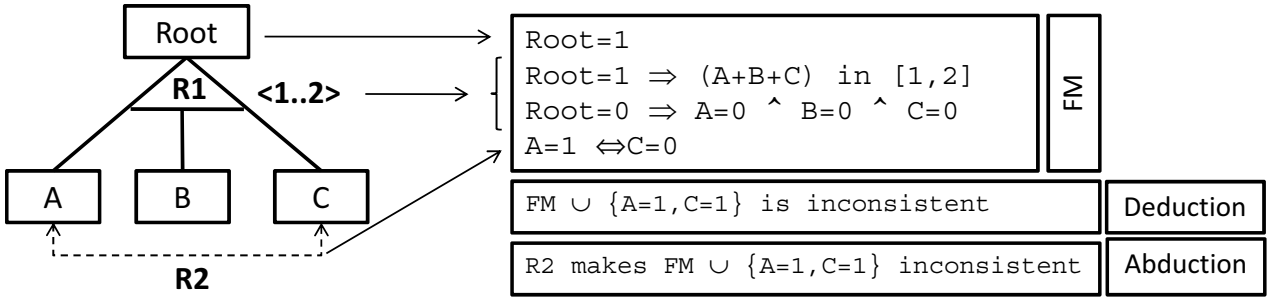


Figure 2. Difference between deductive and abductive reasoning in FM analysis

A more detailed list of deductive operations may be seen in Table 1. All the above operations are deductive ones but explanations and error correction which are abductive operations. Properly speaking, model transformations are not analysis operations as they change the FM so they will be out of our scope. Next Sections we analyse the structure of abductive reasoning and refine the explanation operation to provide into a wider set of abductive operations.

3. Abductive Reasoning in a Nutshell

Most of the applications that use logics commonly use deductive reasoning or deduction. In deductive reasoning we have a conception of our relevant world that is synthesized within a *Knowledge Base*(KB). A KB is composed by a set of facts that are accepted to be true. For example, a FM will be the KB in automated analysis. The objective of deduction is concluding a set of consequences from a KB.

In many contexts, the available information is incomplete or imprecise, normally due to the inability or difficulty of explicitly capturing all the knowledge in a KB. In classical logic, a proposition may be true or false. Anything that is not known or may be inferred is considered to be false in what is called the Closed World Assumption (CWA)[8]. However, when incomplete knowledge appears, we also consider a third state where a proposition is not known to be true or false. Here is where default rules or hypotheses appear. A hypothesis may be considered to be true whenever we have no clue that it is false. However, it makes that a conclusion that we infer from our KB based on hypotheses must be invalidated when new knowledge contradicting the hypothesis appears.

So we need a framework to represent an incomplete knowledge, distinguishing between:

- *Facts* (F): the knowledge that we certainly know to be true. It is a set $\{f_1, \cdots, f_n\}$ of formulas that must be consistent.
- *Default Rules or Hypotheses* (H): A set $\{h_1, \cdots, h_m\}$ of formulas which subsets can be assumed to be true if they are consistent together with the set of facts.

With this structure, for a set of facts and a set of hypothesis, we may have different possible *scenarios* S each of them taking into account a different and valid subset of hypothesis ($S \subseteq H$) consistent with the facts F.

A way to exploit this framework is called *abductive reasoning* or simply *abduction*. The objective of abduction is searching for the scenarios that may explain an observed situation or behaviour in the world. An observed behaviour or *observation* (obs) may be for example a measurement in a physical system or a conclusion obtained using deductive reasoning, and is described as a set of formulas. Rigorously speaking, an scenario S is an *explanation* to an observation obs iff S cannot be entailed to be false from F and $F \cup S$ entails the observation, i.e.

$$F \cup S \models obs$$
$$F \not\models \neg S$$

3.1 Minimalistic Reasoning

From the above definition, we may obtain more than one explanation to an observation so abduction is a non-deterministic problem. In most of the cases, we need a criterion to choose the most suitable explanation and minimalistic reasoning may help on this issue.

Minimalistic reasoning relies on the principle that we normally we are not interested in all the explanations but in the best explanation. To determine the best explanation, we may apply different criteria, but the most typical one is taking the succinctest explanation in what is commonly known as the Occam's razor principle or parsimony law.

Here is where the concept of *minimal explanation* implements the parsimony law. An explanation E is minimal iff for no subset $E' \subset E$, $E\prime$ is an explanation. Therefore, in a problem we will obtain two explanations for an observation $\{h_1, h_2\}$ and $\{h_3\}$ if neither $\{h_1\}$ nor $\{h_2\}$ are able to

explain the observation. It means that $\{h_1, h_2, h_3\}$ may be an explanation but it is removed for the sake of simplicity. A similar but not equivalent criterion to be considered will be choosing the explanations is taking the smallest explanations in terms of the number of hypotheses that are considered. Following this criterion, $\{h_1, h_2\}$ will be removed as an observation since its size is bigger than $\{h_3\}$.

3.2 Diagnosis

A diagnosis problem is one of the main applications of abductive reasoning. Its objective is determining the components that are failing in a system. Diagnosis is widely applied to determine the components that are failing in a circuit and diagnosing diseases in patients from their symptom. To deal with diagnosis problems, one of the most common frameworks is Reiter's Theory of diagnosis[9]. Reiter describes a system in terms of the expected behaviour of its components and how they are linked. Optionally, a description of how a component may fail may be introduced in what is called a *fault model*. Errors are detected by means of observations to the system behaviour and comparing them to its expected behaviour. If expected and real behaviours are different, an error is detected. In other terms, let us represent a system as a set of formulas F and let an observation obs be another set of formulas. An error is detected iff:

$$F \cup obs \models \bot, \text{ or } F \not\models obs$$

Therefore, error may be detected using deductive reasoning, as we are searching for consequences of adding obs to our knowledge. If we intend to go further and explain the reasons why errors happen we face up an abduction problem. As we may observe below, diagnosis problems perfectly fit into the abductive reasoning structure, since:

- The set of facts is the description of the system behaviour, describing both normal and abnormal behaviour of components.
- The set of hypotheses is composed by formulas that represent the normal and abnormal behaviour of each component.
- Observations are provided to obtain explanations to the errors that have been previously detected using deduction.

Therefore, using abductive reasoning we obtain a set of minimal explanations, where an explanation for an error is a list of components that are failing and a list of those that must behave correctly.

Summarizing, a diagnosis problem is an abduction problem where the only available hypothesis are those indicating the normal or abnormal behaviour of components.

3.3 Abduction, Deduction and Automated Analysis

Many operations have been proposed for the AAFM. Most of them are deductive operations since their objective is obtaining conclusions from a logic representation of a FM. However, there is a set of explanatory operations that have been solved using abductive reasoning techniques. As far as we are concerned, there has been no effort to remark this difference. So it is our intention to shed light on the difference between abductive and deductive reasoning so it could be applied in automated analysis.

Figure 3 summarizes our conception of the automated analysis when deductive and abductive operations are distinguished. In deductive operations, we are able to obtain conclusions (or the absence of them) from a FM logical representation that allows deductive reasoning. For abductive operations, we are interested in obtaining explanations from the results or conclusions obtained from a deductive operation. In this case, FMs are represented using logics that distinguish between facts and hypotheses. Deductive and abductive operations use different solvers or reasoners, choosing the most suitable for each kind of operation to be performed.

Next Section, we propose a catalog of abductive operations, and as we will expose later in Section 5, it will be a task of our future work to explain in details the translation of FMs to abductive logics and the solution using different techniques or solvers.

4. Operations Catalog

We present a catalog of operations for the abductive reasoning on FMs. These operations are executed just after a deductive operation. The catalog we present here is inspired by Benavides' [2] catalog of operations. We have selected its deductive operations and some others that have been proposed lately. For each deductive operation, we propose "'why?"' and "'why not?"' abductive questions. "'Why?"' questions are asked when a deductive operation has a solution. "'Why not?"' questions intend to find an answer for a deductive operation that has no solution. Small examples are provided to illustrate their usage.

4.1 Why? questions

A "'Why?"' question intends to explain the result obtained from a deductive operation. It is important to remark that in this case, deductive reasoning is able to obtain a result, but we would also like to know the reason why that result is inferred. We have found four relevant questions of this kind:

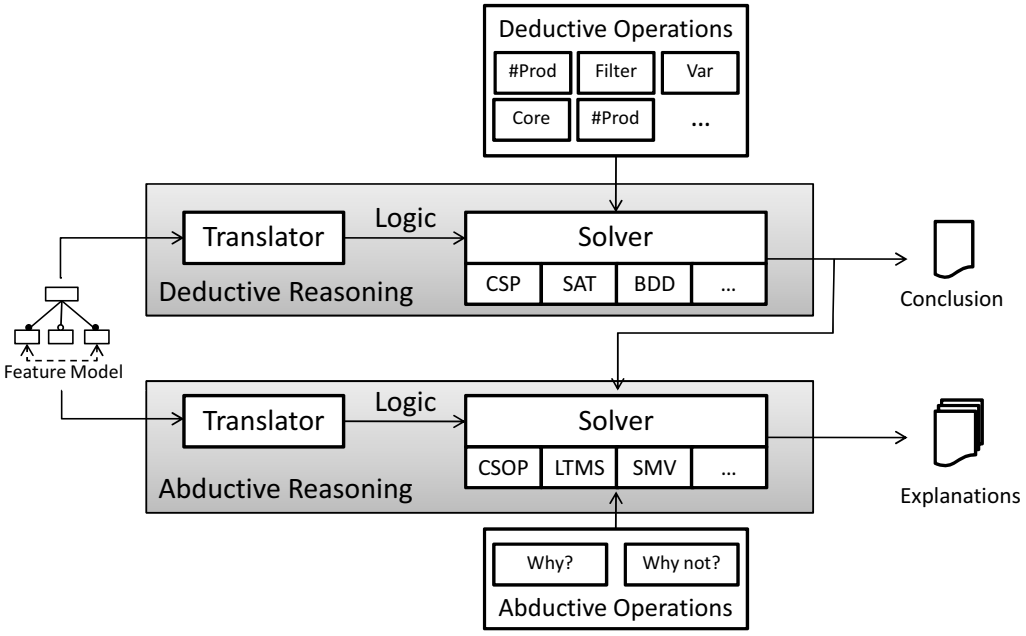
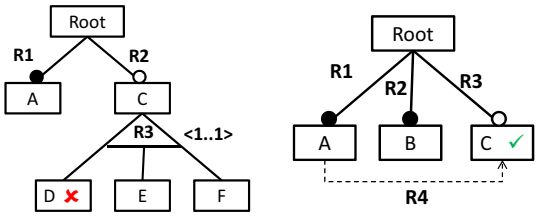


Figure 3. Relating abductive and deductive reasoning to automated analysis of FMs



(a) Operation 1 Example (b) Operation 2 Example

Figure 4. Example Feature Models

Operation 1. Why is it a variant feature? This operation is executed to extend the information obtained from the *"'retrieving the variant features"'* deductive operation. In this scenario, we want to obtain the relationship/s that are becoming a feature variant. Considering the example in Figure 4(a), if we want to determine the relationships that make feature D being variant we have to obtain a justification that concludes that we are able to remove that feature in a configuration. For the example, we will obtain $\{R2\}$ and $\{R3\}$ as two explanations to our question.

Operation 2. Why is it a core feature? The deductive operation *"'Retrieving the core features"'* lists the features that appear in every product or core features. This operation provides the relationships that makes one of those features belong to the core. Considering the example in Figure 4(b), all the features in the FM are core features. We expect C to be a variant feature since it is linked to the root by an optional relationship. *"'Why is it a core feature?"'* operation will highlight $R4$ and $R1$ relationships as a justification for C being a core feature.

We have seen how this operation and the previous one are applied to obtain more information from core and variant features. We must notice that we may also use both of them when we calculate the commonality or variability of a feature. A feature which commonality is 1 is a core feature; if its commonality is not 1 it is a variant feature. Therefore, we may use operation 1 and 2 for these cases.

Operation 3. Why is a partial configuration valid? A partial configuration in a FM is a list of selected and removed features. A complete configuration is a particular case of partial configuration where each feature in the FM is selected or removed. The deductive operation *"'Determining if a configuration is valid"'* infers whether it is possible to select and remove the features in a partial configuration. If a positive response is obtained, we may want to know the relationships that make the partial configuration possible. Let us take the FM in Figure 5 as an example, where the list of selected features is $\{Root, A, C, E\}$

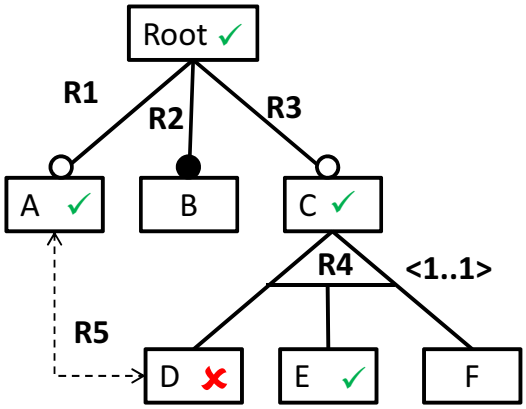


Figure 5. Operation 3 Example

and $\{D\}$ the list of removed features. The result of the abductive operation *"'Why is a partial configuration valid?"'* will return $\{R1, R3, R4, R5\}$ as the set of relationships that affect those features.

Operation 4. Why is a product optimal for a criteria? Finding a product that optimizes a criteria is the objective of the deductive operation *"'Optimizing"'*. This operation is commonly used when extra-functional information is attached to a FM in the so-called extended FMs[3]. In some situations we may be interested in knowing the relationships that have been taken into account to reach a solution. In the example in Figure 4.1, $\{Root, C, E\}$ features form the product that is found to be the cheapest product in the family. The abductive operation *"'Why is a product optimal for a criteria?"'* will obtain $\{R2, R3, cost_{Root}, cost_C, cost_E\}$ as the relationships that make this product optimal. This operation may be seen as a particular case of Operation 3 where the configuration is obtained from an optimization process.

4.2 Why not? questions

Many deductive operations may obtain no solution or a negative response when inconsistencies are found. In the abductive operations that we analyse next, their objective is obtaining further information about the relationships that are making a deductive operation impossible to obtain a solution. As we intend to find the components (relationships in our case) that explain a failure or inconsistent situation, these operations fit into the diagnosis problem, so their results may be used to repair a FM or a configuration.

Operation 5. Why is a feature model not valid? A void FM is the one where it is not possible to derive any

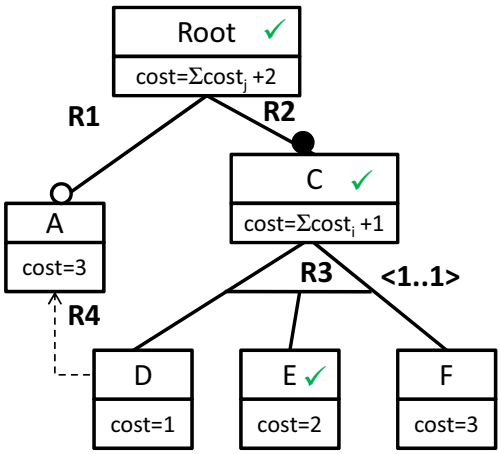
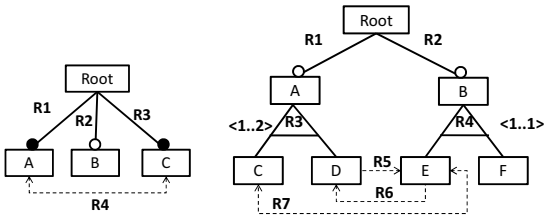


Figure 6. Operation 4 and 11 Example

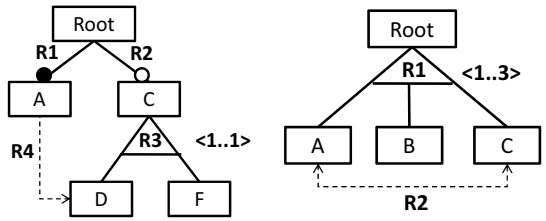


(a) Operation 5 Example (b) Operation 6-7 Example

Figure 7. Example Feature Models 2

product. A FM is valid if it defines at least one product, i.e. it is not void. Void FMs are produced due to contradicting relationships. The deductive operation "'Determining if a FM is void"' tries to find a valid product to demonstrate that a FM is valid. In case it finds no product, the FM is determined to be void and we need to extract information about the relationships that make the FM be void or not valid. *"'Why is a feature model not valid?"'* operation obtains one or more explanations for a void FM, i.e. sets of relationships that prevent the selection of a product. In the example in Figure 7(a), three explanations are obtained: $\{R1\}$,$\{R3\}$ and $\{R4\}$. This information may be used by a feature modeler to correct the FM by relaxing or removing one or more of those relationships.

Operation 6. Why is a product not valid? Whenever the deductive operation *"'Determining if a product is valid"'* detects an invalid product selection, it is mandatory to obtain further information about the relationships that are making the product impossible to



(a) Operation 8 and 9 Example (b) Operation 10 Example

Figure 8. Example Feature Models 3

derive. This operation will be useful when we want to check for a FM to include a set of well-known products. In the example in Figure 7(b), we want the FM to define the product $\{A, B, C, E\}$ (remaining features are supposed to be removed). Deductive operation detects this product as invalid and *"'Why is a product not valid"'* explains this unexpected result by detecting $\{R4, R6\}$ as the relationships that are causing it.

Operation 7. Why is a partial configuration not valid? Whenever *"'Detecting if a configuration is valid"'* detects an invalid configuration, and we know that the configuration must be possible, we may be interested in knowing the relationships that are making it impossible. Taking the FM in Figure 7(b) as example and partial configuration $\{C, D\}$, we obtain relationships $\{R5\}$ and $\{R7\}$ as explanations. From this point of view, we may consider previous operation as a particular case of this one, as a product may be considered as a partial configuration. Another approach to this question would be obtaining the features that must be removed from a configuration if we consider that the FM is correct. In this case, this operation would conclude that feature $\{C\}$ or $\{D\}$ must be removed from the configuration to obtain a valid one.

Operation 8. Why is a feature not selectable (dead feature)? A dead feature is the one that despite of appearing in a FM it cannot be selected for any product. The deductive operation "'Dead features detection"' obtains a list of the dead features in a FM. This operation detects the relationships that are making a dead feature, assisting on the correction of the FM. Taking the FM in Figure 8(a) as example, $\{F\}$ is obtained as the only dead feature in the model. The explanations that we obtain are $\{R1\}$, $\{R3\}$ and $\{R4\}$, one of which must be removed or changed at least to correct the dead feature.

Operation 9. Why is a feature a false-optional? A false-optional (a.k.a. full-mandatory) feature is the one that has an implicit mandatory relationship with its parent feature despite of being linked by an optional relationship. The declarative operation *"'False-optional features detection"'* obtains a list of this kind of features. This abductive operation obtains explanations to repair such an error. In Figure 8(a) example, $\{C, D\}$ features are false-optional, obtaining $\{R1\}$ and $\{R4\}$ as explanations.

Operation 10. Why is a cardinality not selectable (wrong cardinality)? Set-relationships use cardinalities to define the number of child features that may be selected whenever its parent feature is. When a cardinal is never used in any product, we are taking about a wrong cardinality. Although this operation is not theoretically described, it is supported by FAMA Framework [12] which implements a deductive operation *"'wrong-cardinalities detection"'*. Taking Figure 8(b) example, we may notice that it is impossible to select 3 child features since A and C exclude themselves so $R1$ has a wrong cardinality. This operation will provide two explanations $\{R1\}$ and $\{R2\}$ since to correct the error we may remove the cardinality or the "'excludes"' relationship.

Operation 11. Why is there no product following a criteria? When *"'filtering"'* or *"'optimizing"'* deductive operations are unable to find any product, this operation helps on finding the reasons why there is no solution. In the example in Figure 4.1, if we want to find a product which costs less than 4, *"'filtering"'* will obtain no product at all. This operation will provide explanations such as $\{cost_{root}\}$ and $\{cost_E\}$ since they increase the total cost of a product in 2.

4.3 Summary

We present the relations among abductive and deductive operations in Table 1. The list of deductive operations is mainly inspired in [2] and [4] and extended with error analysis operations[11] and configuration operations[15].

In this table, N/A is used to represent those operations that do not fit into abductive reasoning. In this category we place "'determining if two FMs are equivalent"' as it is an operation that compares two FMs and both deductive and abductive reasoning frameworks are only able to deal with just one FM. Corrective explanations are also out of our scope although they are closely connected to explanations. Corrective explanations may be considered as two-step operations where an error is explained firstly and corrected secondly. We are able to provide explanations via abductive reasoning, but suggesting corrections is not so trivial and will be an aim of our future work.

Deductive Operation	Abductive Operations	
	Why? operation	Why not? operation
Determining if a product is valid	N/S	Op.6
Determining if a FM is void	N/S	Op.5
Obtaining all the products	N/S	Op.7
Determining if two FMs are equivalent	N/A	N/A
Retrieving the core features	Op.2	Op.1
Retrieving the variant features	Op.1	Op.2
Calculating the number of products	N/S	Op.5
Calculating variability	Op.1 or Op.2	Op.8
Calculating commonality	Op.1 or Op.2	Op.8
Filtering a set of products	N/S	Op.6,7,11
Optimizing	Op.4	Op.11
Dead features detection	N/S	Op.8
Proving Explanations[1]	Op.1-4	Op.5-11
Providing Corrective Explanations	N/S	N/A
False-optional features detection	N/S	Op.9
Wrong-cardinalities detection	N/S	Op.10
Determining if a configuration is valid	Op.3	Op.7

[1] All the operations described in the table provide explanations for different contexts

Table 1. Relation between deductive and abductive operations

N/S is used to remark the operations that could be performed but will have no sense from the point of view of the automated analysis. For example, we are not interested in determining why a FM describes 20 products. However we must be interested in knowing why there a FM describes no product.

5. Conclusions and Future Work

In this work, we have presented our conception of AAFM from the point of view of the kind of reasoning needed to solve the different analysis operations. We have presented a new catalog of operations that rely on abductive reasoning and some of which have already been dealt with in some previous works, but the remaining operations are new. As a first step in our roadmap of integrating abduction in AAFM it is our intention to open a debate where the proposed catalogue of abductive operations is extended or reduced.

Once we have obtained a stable catalogue, we envision that we need two main pieces to complete the puzzle of abductive reasoning:

1. A translation from FMs to non monotonic logics, i.e. logics that are able to represent incomplete knowledge.

2. A solver-independent solution to all the abductive operations so that different solvers can be used to execute these operations.

We will implement the solutions to these operations into FAMA Framework [12]. Currently FAMA Framework supports explanations for operations 7 to 10 by means of CSOP that you may download at `www.isa.us.es/fama`. After obtaining these results, we will design benchmarks to analysing the solvers that perform better for each abductive operation.

References

[1] D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference, LNCS 3714*, pages 7–20, 2005.

[2] D. Benavides. *On the Automated Analysis of Software Product Lines Using Feature Models. A framework for developing automated tool support.* PhD thesis, University of Seville, http://www.lsi.us.es/~dbc/dbc_archivos/pubs/benavides07-phd.pdf, 2007.

[3] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005.

[4] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feture models. In *Jornadas de Ingeniera del Software y Bases de Datos (JISBD)*, 2006.

[5] K. Czarnecki and P. Kim. Cardinality-based feature modeling and constraints: A progress report. In *Proceedings of the International Workshop on Software Factories At OOPSLA 2005*, 2005.

[6] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature–Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Nov. 1990.

[7] M. Mannion. Using First-Order Logic for Product Line Model Validation. In *Proceedings of the Second Software Product Line Conference (SPLC2)*, LNCS 2379, pages 176–187, San Diego, CA, 2002. Springer.

[8] R. Reiter. On closed world data bases. pages 300–310, 1987.

[9] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[10] J. Sun, H. Zhang, Y. Li, and H. Wang. Formal semantics and verification for feature modeling. In *Proceedings of the ICECSS05*, 2005.

[11] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6):883–896, 2008.

[12] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A.Jimenez. Fama framework. In *12th Software Product Lines Conference (SPLC)*, 2008.

[13] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and M. Toro. Explanations for agile feature models. In *Procceedings of the 1st International Workshop on Agile Product Line Engineering (APLE'06)*.

[14] H. Wang, Y. Li, J. Sun, H. Zhang, and J. Pan. A semantic web approach to feature modeling and verification. In *Workshop on Semantic Web Enabled Software Engineering (SWESE'05)*, November 2005.

[15] J. White, D. Schmidt, D. B. P. Trinidad, and A. Ruiz-Cortes. Automated diagnosis of product-line configuration errors in feature models. In *Proceedings of the Sofware Product Line Conference*, 2008.

An Industrial Case Study on Large-Scale Variability Management for Product Configuration in the Mobile Handset Domain

Krzysztof Wnuk, Björn Regnell, Jonas Andersson and Samuel Nygren
Dept. of Computer Science, Lund University, Sweden
{krzysztof.wnuk, bjorn.regnell}@cs.lth.se

Abstract

Efficient variability management is a key issue in large-scale product line engineering, where products with different propositions are built on a common platform. Variability management implies challenges both on requirements engineering and configuration management. This paper presents findings from an improvement effort in an industrial case study including the following contributions: problem statements based on an interview study of current practice, an improvement proposal that addresses the challenges found, and an initial validation of the proposal based on interviews with experts from the case company.

1. Introduction

Software Product Lines have already proven to be a successful approach in providing a strategic reuse of assets within an organization [9]. In this context, variability management is considered as one of the key for successful product lines and concerns in all phases of the software product line lifecycle [8]. We experience considerable growth of the amount of variability that has to be managed and supported in software assets. Inspired by the previous fact, we have conducted an industrial case study focusing on the process of variability management at one of our industrial partners in the mobile phone domain. The topic of our investigation was an established product line engineering process [9] in a company that sells over 50 products every year worldwide in millions of exemplars. Our goal for this study is to increase the knowledge of how the products are configured by studying current issues and if possible proposing and evaluating improvements. To address the goal we have formulated three research questions:

Q1: How are variability requirements and variability points managed in software product lines in practice?
Q2: What are the problems with managing variability requirements and product derivation?
Q3: What improvements can be made in managing variability?

The first two questions were addressed by an interview study, were we have investigated the process of *product derivation* [7] and the concept of *managed variability* [9]. By using managed variability we refer to defining and exploiting variability throughout the different life cycle stages of a software product line [9]. In total 29 persons working with requirements engineering, implementation and testing were interviewed in order to understand how the variability is represented, implemented, specified and bound during the product configuration. As a result, a set of challenges is defined and presented in this paper.

To address Q3, we have proposed and evaluated improvements to the current way of working. Our main proposal includes a new structure of variability information that aims at enable linking product configuration to the initial requirements. It includes splitting the configuration into two levels of granularity. Additionally, we propose to use a main product specification with entities that can be consistently applied throughout the whole organization and will address current documentation issues.

Finally, we have empirically evaluated our improvement proposals by applying them to the existing configuration structure in a pilot study. Additionally, we have conducted a survey by sending questionnaires about the potential benefits and drawbacks of our proposal. 28 out of 34 persons have answered our questionnaire. Most of the respondents expressed positive opinions about the proposal and did not express any major obstacles that may apply to it.

The reminder of this paper is organized as follows. In section 2, we describe the industrial context of the

case study. In section 3, we provide a description of research methodology. In section 4, we discuss identified problems and issues. In section 5, we describe improvement proposals, which we evaluate in section 6. Section 7 presents related work and the paper is concluded in Section 8.

2. Industrial Context

The case study was performed at the company that has more than 5 000 employees and develops embedded systems for a global market. The company is using a product line approach [9]. Each product line covers different technologies and markets. The variability of the software product lines in our case are organized in two dimensions. The first dimension represents product segments or product technologies, and the second represents the code base that evolves over time. In each of the clusters there is one lead product built from the platform representing most of the platform functionality. The lead product is scaled down to create sub-products and new variants for other markets and customers. Some of the sub-products originating from the main product contain new features [9]. The platform development process is separated from the product development process as described by Deelstra et. al in [7].

Organization. There are three groups of specialists working with the requirements part of the platform project: *Requirements Engineers*, *Requirements Coordinators* and *Product Requirements Coordinators*. Each technical area in the products domain has a requirements engineers group responsible for covering the progress in the focused field. Their involvement in the projects is mainly focused on the platform project where they supply high level requirements derived from roadmaps, product concepts and customer requirements. They are also main responsible for the scoping process of the platform. Requirements coordinators work between requirements engineers and developers. Their main role is to communicate requirements to the developers and assist with creating detailed design documents and requirements. Product requirements coordinators are responsible for the communication of the requirements between the product planner and requirements engineers on the specific product level.

The *Development Teams* are responsible for implementing the software in the platform. They review the requirements and estimate the effort needed for implementation. Each new functionality is assigned to a primary development team which is responsible for its implementation in the software modules. Newly implemented functionality is later tested before final delivery to the platform. The different modules need to be integrated and compiled to a full system. This stage is done by the *Product Configuration Managers (PCMs)* team which manages the different variants and versions of the products created from the platform. The compiled system is tested by a product focused testing organization, *Product Software Verification.*

Requirements Management Process. The company is using two types of containers to bundle requirements for different purposes: *Features* and *Configuration Packages (CPs).* As a *feature* we consider in this case a bundle of requirements that we can estimate market value and implementation effort and use those values later in the project scoping and prioritization. Configuration packages are used to differentiate the products by selecting different packages for different products. The company is using the similar approach to CPs as described in [10], where a configuration package is a set of requirements grouped to form a logical unit of functionality. Every requirement has to be associated with one or more CPs. The requirements engineers list the changes and CPs in their area of expertise in the *Configuration Package Module*. These modules have dependencies between each other and some of them are mutually exclusive [10]. CPs that are common for all products in a product line are marked with an attribute stating that these packages cannot be removed from a product configuration. Hardware dependencies, which make individual requirements valid or invalid for different products, are also specified by the use of *Configuration Dependencies* on the requirements level. The model is similar to the Orthogonal Variability Model proposed by Pohl et al [9].

Product Planning. *Product Planners* are responsible for defining products from the platform available in a product line. They belong to the marketing division in the company so their task is to create an attractive product offer [3] rather than to perform the actual configuration of it. The product planers establish a concept of a new product which induces commercial overview, price range, competitor analysis and gives an overview of the high level requirements. This document serves as a basis for the *Product Configuration Specification,* which specifies the product based on capabilities offered by the platform. The product configuration specification specifies the configuration of a product concerning both software and hardware using the configuration packages defined in the configuration package modules including

configuration dependencies. This model is also similar to the Orthogonal Variability Model proposed by Pohl et al [9]. The product configuration specification corresponds to the application variability model of the Orthogonal Variability Model.

Product Configuration Management. Product Configuration Management teams are responsible for integrating, building and managing variants in the product line. When configuring a new product from the product line, the product configuration manager uses hardware constraints derived from a hardware specification for each product in a cluster to set and configure the software. At this stage, the traceability from the configuration parameters to the requirements is crucial. This part of the context is the subject for the improvement proposal in section 5.

3. Research Methodology

In order to get a comprehensive picture of how variability management is performed at our case company, we decided to conduct a set of interviews with various employees in various positions within the company. The requirements management tool architecture was also explored to understand how variability is defined at the requirement level. During this phase the persons involved in process improvement for the requirements process were interviewed and consulted with during the exploration of the requirements management process.

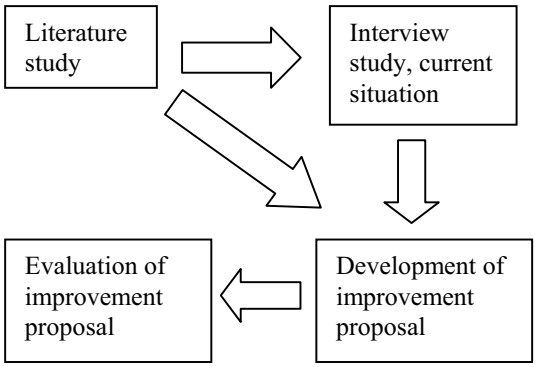


Figure 1. Research methodology.

The next step was to select key personnel to interview in order to get as many different perspectives how variability is managed and how products are configured as possible. By analyzing the case company's product configuration interface, the amount of variation for different development groups was established. One group with a large amount of product variations and one group with a small amount were selected for further investigation. To cover the whole process of variability, we have involved Product Planners, Requirements Engineers, Requirements Coordinators, Developers and System Testers in our study.

The interviewed persons were selected based on their position in the company. Some persons were recommended by already interviewed. In some cases the person that was asked to participate in our study suggested a colleague as a replacement with the motivation that he was more familiar with the area. In total, 27 persons were interviewed. The interviews were semi-structured in order to allow the interview to change direction depending on the interviewee's answer, and adapted for the different roles and the progress of the interview study. This approach balances between early interviews that were more focused on the general aspects with later more specific interviews. The interviews took approximately one hour. During this time interviewers took notes continuously which were later summarized. During summarization, discrepancies between interviewers interpretation were discussed and, if needed, formulated as questions that were later sent to the interviewee. Apart from the summary, the interviewee also received a model of how he or she perceived the process of variability management. After interviewee approval, which sometimes was done after some minor changes, the data was ready to be analyzed. After interviewing 27 persons, it was decided that the received overview of the current process was satisfactory to proceed with analysis and propose improvements. Sample questions used at the interviews and distribution of interviewed personnel can be accessed at [15].

4. Results

In this section we present the results from our interview study. We describe the different perspectives on the configuration process, configuration activity measurements, and finally the problems that were identified.

4.1 Perspectives on the Configuration Process

Most of the stakeholders have a common view of how products are created. The product projects create a product concept, which is then used by requirements engineers in defining platform requirements. Later in the process the product planners are involved in creation and configuration of new products by creating

change requests issues regarding both new and existing functionality. When previously created formal change request is accepted, it is send to the assigned developers team which performs implementation or configuration changes. The differentiation achieved in this manner is not explicitly documented in product specification but only in the minutes from the change board meetings. In the next section, the deviation from this common view is described, as well as the differences from the documented process model.

Product requirements coordinators, requirements coordinators and requirements engineers have limited knowledge about how variability is achieved due to their focus on the platform. They also state that developers do receive most of the configuration instructions through bug report issues from product planners, customer responsible and testers. We discovered that some variability is stated in the requirements' text in an implicit way creating problems with recognition and interpretation at the development phase. Product planners' knowledge about configuration packages is limited and they have not experienced the need for a better product documentation than what is delivered in the concept definition.

The developers express the opinion that information regarding variability is not communicated in a formal way. Instead, they get information about variability through their team leaders in a form of change requests at the late stages of development. These change requests are often used to configure products. The creation of new variation points is done in the platform project, and is therefore often based on assumptions made by the developers out of the previous experiences and informal communication with people involved in the process. The main opinion is that the information about what value that should be assigned to a variation point is possessed by individuals. The information is also not documented sufficiently in formal documents. Requests for new variation points or values are forwarded to the product configuration managers.

Product Configuration Management Perspective. We discovered that the product derivation process is iterative and similar to the one described by Deelsta et al [7]. When a main product for a product line is created from the platform, it is based on the existing configuration of the previous similar product. This configuration is adjusted to the new hardware specification for the platform. Since the amount of configuration parameters in the configuration file has increased significantly, and they are not sufficiently documented product configuration managers are unable to keep track of all changes.

When a new product has been set up, it is built and sent to the product testers. Their task is to test the product and to try to discover software errors and functionality that might be missing. At this stage it is often difficult for the testers to determine whether errors depend on faulty configuration or software errors. Therefore they create a bug report towards the developers to initiate investigation of the reason of the failure. The errors are corrected by developers and new source code is later sent back to the product configuration manager, which is merging the delivered code from all development groups.

When the sub-product is created, the most similar product configuration is copied from the previous products. Next, the configuration manager responsible for the sub-products is trying to configure the product by checking product structure documentation and other relevant information. The required information is gained from multiple sources, which leads to the double maintenance problem described by Babich [11], where uncertainties about the values of variation points are concluded by comparing with other projects. As a result a time consuming investigations have to be perform and very often influences the speed and correctness of the product configuration.

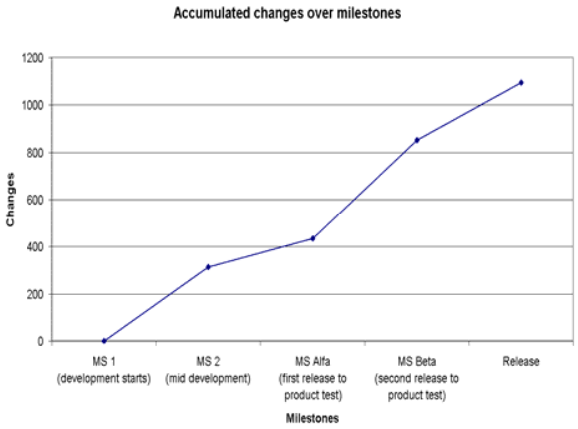


Figure 2. Accumulated changes to the configuration over milestones.

4.2. Configuration Activity Measurements

In order to understand how the configuration is changed over time, change related measurements were defined. The configuration file was chosen for each label of the code base in the product line. Labels are used to tag revisions of files produced by developers that are to be used by product configuration manager. The differences between each configuration file were

calculated in order to get measurements describing how many parameters that were added, deleted or changed. The results are visualized in figures 2 and 3. Note that over 60% of the configuration changes are done after the software has been shipped to the testers (MS Alfa).

The results support our previous observations derived from interviews, where developers admit that they configure the products based on bug reports and change requests. At the time this study was performed, the configuration had over one thousand different parameters available at the product level, spread across a configuration file of thousands of lines. These parameters were controlling over 30 000 variation points in the source code with different levels of granularity. Further analysis showed, that one configuration parameter controls an average of 28 variations points, which suggests that most of the

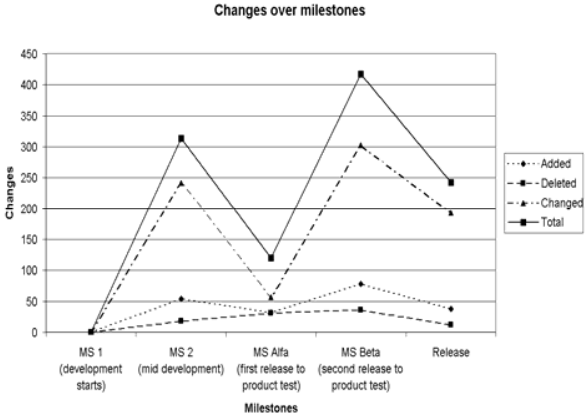


Figure 3. Changes to the configuration over milestones.

variability is quite fragmented. The source code consists of millions of lines of code in more than 10 000 files, giving an average 250 lines of code per variation point.

4.3. Problems Identified

According to Van Der Linden et al [3], the configuration manager should be responsible for maintaining the configuration of all variants and ensuring that the functionality for all products is covered. In our case it remains unclear who is responsible for binding the variation points of the platform to create a specific products. As a result, we experience creation of variation point that have no specific owner. Furthermore, since most of the development and architectural activities are platform focused and a role such as Application Architect or Product Architect responsible for binding variation points of the platform to create specific products is not present in the current organization [9]. The lack of clear responsibilities results in an absence of clear, specific and strategic goals and long term improvements.

The configuration of new products is achieved in an iterative manner between developers, configuration management and testers [7]. Due to the lack of a specific ownership, the configuration is not always properly reviewed, which is often a reason for missing functionality. As a result, testing and maintenance efforts may increase. The knowledge about product derivation and variability is not formalized [7,10].

As mentioned previously, the unrestricted rules for creating and managing variation points results in their excessive creation. Many variation points become obsolete either due to the fact that they were not created for product configuration purposes or because of the complex dependencies. It is undefined who is responsible for removing these obsolete variation points from the configuration file. This fact makes the configuration file hard to manage and overview.

In our case, the flexibility that needs to be copied by standardization of the product line [9], in the sense of amount of variation points is too great and offers many more configuration capabilities than is needed for product configuration and differentiation. The number of variation points, and their structure is too complex to be managed by the people responsible for the product configuration and differentiation. The variability capabilities need to be more standardized and less detailed to handle the costs associated with the flexibility.

The biggest challenge throughout the organization turned out to be the lack of complete product specifications, which may lead to the following problems:

- Time consuming “detective” work where information is gathered through informal communication and unofficial documents.
- Faulty bug reports.
- Double maintenance of fragmented product information that exists in different documents and versions throughout the organization.
- Faulty configuration.
- Critical knowledge about variability configuring products possessed by individuals.
- Increased effort in verifying the configuration of a product.

These problems is tackled by the use of unofficial documents specifying the product characteristics for

both hardware and software. The documents are created in an informal way and are neither reviewed nor a part of the formal approval process, but still used throughout the organization. These documents and the related process can be improved with respect to configuration management, as uncontrolled documentation procedures may result in unintended product configurations.

5. Improvement Proposal

In order to improve the issues presented in section 4.3, we have developed a set of improvements regarding variability documentation, granularity and management.

Improved traceability between requirements and variants. Our proposal will reuse the configuration package concept, described in section 2, to associate the configuration parameters with the requirements. The configuration packages should be used by the product planners to configure the products. By associating the configuration packages with the configuration parameters, traceability links to both requirements and configuration parameters will be established. The division into configuration packages should be done in cooperation between developers and requirements engineers to fully capture all possible aspects of variability. Newly created variation points should be explicitly documented and spread across all stakeholders. This approach will result in a more complete traceability between the configuration packages and the configuration interface, and can be a step towards the automatic generation of a product configuration directly from the product configuration specification in the future.

Abstraction layer. The overview of the proposed abstraction level is described in figure 4. In the current structure the configuration file contains all detailed feature configuration on a very low level for all products defined. The file is edited by both product configuration managers and developers and because of its size and granularity it is vulnerable and subject to merge conflicts. Our proposal introduces a new abstraction layer, *CP-Conf,* between the product configuration interface and the software modules. The low level configuration is moved into the lower layer, and a high level product configuration based on the configuration packages is used on the product configuration level. In this solution, the developers are becoming responsible for the *CP-Conf* layer and the modules associated with it. The product configuration manager is only responsible for the high level product configuration. To be able to introduce an abstraction level, configuration

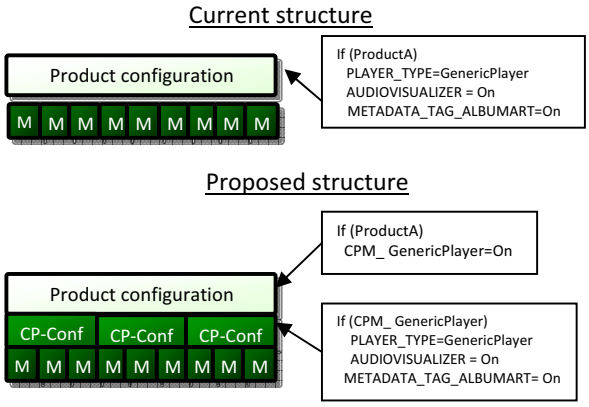


Figure 4. Overview of the proposed abstraction layer.

parameters in the configuration file need to be moved to a separated files where a parameters belonging to a certain development team reside. The specification of selected modules needs to be in these separated files too, since it depends on the selected configuration packages. Also, when this abstraction layer is introduced and the parameters are named according to the configuration packages, there should be no need to change the existing variation point naming since the parameters will be moved out from the main configuration file. The solution is described in figure 5.

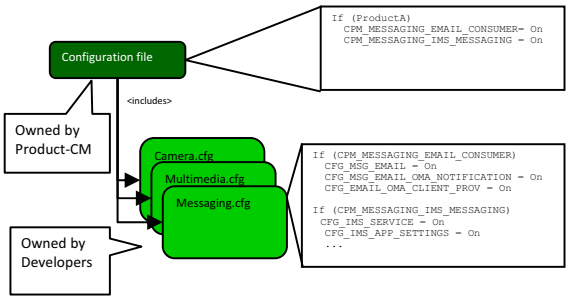


Figure 5. Configuration is distributed into configuration files according to the concept of Configuration Packages.

New configuration parameters. Today the naming of the configuration parameters includes a feature description indicating what functionality the parameter affects. However, the features in the configuration parameters are not mapped to the requirements by including an identifier connected to a specific

requirement. Since the feature names originate from two sources, traceability is based only on human reasoning. We propose a new standard for configuration parameters where four types of parameters are available:

- The existing low level parameters which are presently used for product configuration. To remove or change these parameters is an infeasible work.
- The existing parameters which define the hardware properties of the product should be assigned a prefix CFG_HW. Today many of the parameters created are hardware dependent and could therefore be removed by using the hardware properties instead of creating new parameters. The syntax of the parameters should include the serial number from the hardware requirements specifying its value.
- A new type of parameter for configuration dependencies. The name should include the dependency type (HW/FormFactor/ Customer/Market). The syntax can e.g. be CD_<TYPE>_<NAME>.
- An internal binding should be used when software varies non-significantly.

Documenting variability. Currently, the documentation of variation points is not mandatory and resulting in its incompleteness. Since developers in our proposal will be responsible for the lower levels of variability, the documentation process will be simplified by responsible stakeholders' constraining. By introducing traceability between the product level configuration interface and the configuration packages, no further documentation is needed on the higher level. The name standard will be descriptive and in line with the configuration packages. It will enable stakeholders to find more information in the requirements management system, where the configuration packages are defined, described and associated with requirements.

Managing obsolete configurations. Many parameters in the configuration file are obsolete. Because of that we propose that the configuration file should be locked for changes. Parameters that do change but have the same value for all products should be moved to the development team's specific file, and should not be a part of any configuration package. Similar to the configuration parameters, obsolete configuration packages that are not used in any product should be moved out from the software product line. If a configuration package is used in any product it should be incorporated into the platform and removed from the set of CPs. In the same fashion as the configuration packages, the high level hardware parameters should be left at the product configuration level, while its associated low level parameters should be moved to the proposed low abstraction layer and owned by the developers.

Availability of product specifications. All available configuration packages in the platform should be included in the product configuration specification, and a connection to the previously mentioned abstraction layer should be made. By applying this approach, the task of configuring a new product will be simplified and could possibly be automated in the future. The automatic configuration file generation can be based on the configuration packages defined in the requirements management tool.

6. Evaluation of the Proposals

The evaluation of the implemented proposals was carried out as a desktop pilot [12], where the new structure was applied to the existing structure. The desktop pilot was run on a subset of the configurations belonging to two development teams. Two developers from each team, chosen based on their knowledge about configuration parameters, have participated in the redefinition part of the evaluation. The configuration packages defined by requirements engineers were used to group the existing low level configuration parameters, as described in the proposal. This was done in cooperation with the developers. When parameters could not be linked to a certain existing configuration package, the developers had to consider defining a new configuration package, configuration dependencies or hardware requirements. From these lessons learned we can conclude that:

- Packages need to be complemented with a more complex version for greater differentiation possibilities
- Some packages need to have defined dependencies to other packages
- The differences between some of the similar configuration packages need to be described by requirements engineers
- One package may in the future need to be split into several packages that contain end-user functionality and one common package that does not offer any end-user benefits. This one package is dependent on others previously described.
- Problems may arise when new configuration packages need to be created instantly. In this

case the bottleneck will be the communication with requirements engineers.
- There are packages that can be removed from the product due to strong dependencies. In this case, product planners should not be allowed to deselect these packages.

After the redefinition of the configuration, the developers were asked to fill in the evaluation form [13], answering questions concerning the improvement proposal and its possible benefits and drawbacks. To get as many answers as possible, the information was held short and concise. The evaluation form was also sent out to all members in the first development group and to half of the members in the second group, totaling with 34 persons. 28 out of 34 persons have answered and the detailed results are accessible in [14].

From the evaluation it can be seen that the participants have been involved in the product configuration. They also see problems with how it is handled today. The proposal was considered as easy to understand and implement.

Some responders mentioned that customer specifications were not addressed enough. One participant also addressed a need for training in variability management. Most of the participants thought that the responsibilities and the separation of product and feature configuration is easy to understand. In the qualitative part of the results, it was confirmed that the workload will be reduced by improved division of responsibilities.

Most responders strongly agreed to that our proposal should increase the quality of products. On the other hand, a few responders claimed that the quality of the products is now high enough and that our proposal will not make any significant difference. The question addressing improvement in the configuration efficiency scored above average, which indicates that this proposal would have a significant effect on efficiency in the way of working rather than end-product quality. This was emphasized by some people who stated that the configuration would become more manageable and less time consuming.

On the question regarding drawbacks there were concerns that the configuration packages may get too large and fail to offer the needed from market perspective detailed level of configuration. It was also mentioned that there will be a stabilization period until the CPs are clearly defined. One responder expects that quick fixes will be hard to handle using CPs, and that there therefore could lead to the *"quick and dirty"* solutions which are hard to maintain. There is a risk that the number of CPs will increase and that the same problems will arise again. Some responders were also worried about customer specific configurations, which the proposal does not specify in detail. Most participants stated that their work will not be affected negatively. Moreover, they stated that there will be less work for the developers with the proposal. The developers would have fewer responsibilities and for some participants their responsibility for product configuration will be completely removed. Overall, the proposal was considered as a better solution than the current way of working.

In the evaluation with the configuration management strategists the responses were positive. Among the positive comments were the possibilities to define a clear process with unambiguous responsibilities, to automate product derivation and verification and to improve the product derivation process. The concerns regarded the need for a streamlined process for managing the configuration packages, including exception handling. Possible dependency problems when a configuration package spans many development teams were also discussed. The overall impression was very positive.

Threats to validity. The way how people were chosen to participate in the interviews can lead to insufficient results. By getting recommendations to which people to interview the risk of getting a subjective picture increases.

The results can be biased by continuous communication with the contact person in the company or by the fact that some concerned stakeholders might have been overlooked in different parts of the case study.

When performing these kind of evaluations, it is difficult to cover all aspects. We are aware that this evaluation only takes a few of the affected development teams into account, and therefore some important information may not be reached. Furthermore, the amount of variation points that each development team is responsible for or shares with other groups varies. Therefore, the scale of affection of the proposal on each development team may vary.

We have not yet performed any evaluation among other stakeholders, like product planning and requirements engineers. Although they are not involved in the technical parts of the proposal, they are part of the process associated with the proposal and it is therefore a drawback not to have these stakeholders represented in the evaluation.

We also see some challenges concerning the ability to maintain the new way of working. It is important that the configuration packages only reflect the current needs for variability and that the configuration packages are not created proactively in the same

manner as variation points are created today. It is also important to educate people in order to consistently convince them of the gains achieved about the new praxis.

7. Related empirical work

Industrial case studies in existing literature [1,2,3,4,5,7] describe the process of introducing product lines. These studies report similar problems to those reported in this paper appear. For example, in the Thales case [7] documentation of the platform has deviated from the actual functionality as the platform has evolved. In other cases [1,4] the enormous amount of low level variability in software was reported. Clements et. al [5] reported that the variability was present only on the files and folders level. In the Philips case [3], the problem of too many dependencies between components, resulting in much time spent on integration problems, was reported. Patzke et. al [6] discovered that many of the differentiation point were actually obsolete and not used any more. The company was also struggling with outdated documentation that was not updated regularly.

In most cases a product line approach was introduced in an evolutionary way, apart from one example [4], where all ongoing projects were paused and the resources were moved to the introduction of the product line project. In some cases, the product line was developed around a new architecture, while assets were derived from an existing base e.g. [3]. Sometimes, a new product line was based on the most similar product from the most recent project. Some cases, like [1], claim that their main success was achieved in the architecture and reorganization, and resulted in the change of the hardware to software cost ratio from 35:65 to 80:20.

The issue of improved traceability between requirements models and variant has been addressed in the literature. For example, Clotet *et al.* [16] present an approach that integrates goal-oriented models and variability models while Alfarez *et al.* [17] present a traceability meta-model between features and use cases. Both example cases seem to be domain independent but are evaluated on relatively small examples which leaves the question of applicability in a large-scale industrial context open.

8. Conclusions

As mentioned in introduction, software product lines improves the quality of the products and reduces the time spent on a product development. However, managing a product line and its variation points efficiently requires a consistent way of working and clear responsibilities. In this case study it has been found that new products are derived by copying the most similar configuration from previous products and iteratively configuring the product between developers, CM and testers. The variability is neither clearly specified nor documented. The responsibilities are unclear. There is no connection between the requirements and the configuration possibilities in the product line. These aspects affect negatively the possibilities to verify the configuration and the time spent on product configuration.

To be able to cope with these issues, improvement consisting of an abstraction layer in the configuration interface have been proposed. This abstraction separates the low level feature configuration from the high level product configuration, and establishes a traceability from requirements to configuration. To clarify the product configuration and ensure that everyone is working consistently, we propose that a product specification, based on these configuration packages, is used throughout the company. Below, we summarize identified problems and corresponding possible improvements:

- *Large number of variation points with an unmanageable granularity.* Variation points are encapsulated into configuration packages, separating the high level configuration from the low level configuration, and resolving the granularity issues.
- *Unclear responsibilities and unstable process for the product configuration.* By dividing the configuration into different layers and proposing responsibilities are clarified.
- *No clear traceability between configuration parameters and initial requirements*. By introducing an abstraction layer based on configuration packages, the configurations are directly linked to the initial requirements.
- *No complete product specification available*. A new and managed product specification based on configuration packages are spread throughout the organization and used by all stakeholders.
- *Products are configured in an inefficient and iterative process without using the initial requirements.* By the use of a complete product specification and a configuration interface based on the same configuration packages, the configuration can be done at early stage.

The evaluation of our proposal shows that the developers are coherently positive to the suggested improvements. To validate out proposals, the changes were simulated together with two development teams. The results showed no major obstacles, but emphasized the importance of cooperation between the requirements engineers and the developers in the definition of the configuration packages. The expectations of this proposal are as follows:

- to reduce effort and time spent on iterative configuration,
- to ensure a higher product quality by improved product verification,
- to state more clear responsibilities among stakeholders,
- to make the information concerning variability within the company more accessible.

It is stated in [9] that explicit documentation of variability can help to improve making decisions, communication and traceability. Following [9] we can also conclude that introducing abstraction levels for variation points and variants improves understanding and management of software product line variability. As a result, we conclude, that our improvement proposals may be relevant for other contexts by addressing the general issue of variability in software product lines with abstraction mechanisms on both requirements and realization level [8].

This paper contributes in a detailed investigation on product derivation from a large software product line, which addresses research question Q1. Question 2 is addressed in section 4.3 as a set of challenges in practice. Finally, Q3 is addressed by the improvement proposals, decribed in section 5 that may increasing product quality and decreasing the effort needed for product defiviation.

Acknowledgements. This work is supported by VINNOVA (Swedish Agency for Innovation Systems) within the UPITER project. Special acknowledgements to Per Åsfält for valuable contributions on problem statements and research direction.

9. References

[1] L. Brownsword and P. Clements, "A Case Study in Successful Product Line Development", Technical Report no. CMU/SEI-96-TR-016, Carnegie–Mellon Software Engineering Institute, Pittsburgh USA, 1996.

[2] A. Jaaksi, "Developing mobile browsers in a product line", *IEEE Software,* IEEE Computer Society, 2002, pp. 73-80.

[3] Linden, Frank J., K.van der Schmid and E. Rommes, *Software Product Lines in Action The Best Industrial Practice in Product Line Engineering,* Springer-Verlag, Berlin Heidelberg, 2007.

[4] Clements P. and L. Northrop, *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional, 2002.

[5] Clements P. and L. Northrop, *Salion, Inc.: A Software Product Line Case Study,* Technical Report CMU/SEI-2002-TR-038, Carnegie Mellon Software Engineering Institute, Pittsburg , 2002.

[6] T. Patzke, R. Kolb, D. Muthig and K. Yamauchi, "Refactoring a legacy component for reuse in a software product line: a case study", *Journal of Software Maintenance and Evolution: Research and Practice*, John Wiley & Sons, UK, 2006, pp.109-132.

[7] S. Deelstra, M. Sinnena and J. Bosch, "Product Deriviation in software product families: a case study", *The Journal of Systems and Software,* Elsevier, New York USA, 2000, pp.173-194.

[8] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. H. Obbink, K. Pohl, "Variability Issues in Software Product Lines", Software Product-Family Engineering. 4th International Workshop, Springer-Verlag, Bilbao, Spain, 3-5 Oct. 2001, pp. 13-21.

[9] Pohl, C., G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques,* Springer-Verlag, New York USA, 2005.

[10] Bosch J., *Design and Use of Software Architectures Adopting and evolving a product-line approach*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[11] Babich, W.A, *Software configuration management: coordination for team productivity.* Addison-Wesley Longman Publishing Co.,Inc., Boston, MA USA, 1986.

[12] R. L. Glass, "Pilot Studies: What, Why and How", *Journal of Systems and Software,* Elsevier Science Inc, New York USA, 1997, pp. 85-97.

[13] Evaluation form can be accessed at http://www.cs.lth.se/home/Krzysztof_Wnuk/VaMoS_2009/EvaluationForms.pdf

[14] Results of evaluation can be accessed at http://www.cs.lth.se/home/Krzysztof_Wnuk/VaMoS_2009/ResultsOfTheEvaluation.pdf

[15] The interview's instrument and participants distribution can be accessed at http://www.cs.lth.se/home/Krzysztof_Wnuk/VaMoS_2009/InterviewInstumentAndDistribution.pdf

[16] R. Clotet, D. Dhungana, X. Franch, P. Grunbacher, L. Lopez, J. Marco and N. Seyff, "Dealing with Changes in Service-Oriented Computing Through Integrated Goal and Variability Modelling", Second International Workshop on Variability Modelling of Software-intensive Systems, Universität Duisburg-Essen, Germany, 2008, pp.43-52.

[17] M. Alfarez, U. Kulesza, A. Moreira, J. Araujo, V. Amaral, "Tracing between Features and Use Cases: A Model-Driven Approach", Second International Workshop on Variability Modelling of Software-intensive Systems, Universität Duisburg-Essen, Germany, 2008, pp.81-88.

A Design of a Configurable Feature Model Configurator*

Goetz Botterweck
Lero
University of Limerick
goetz.botterweck@lero.ie

Mikoláš Janota
Lero
University College Dublin
mikolas.janota@ucd.ie

Denny Schneeweiss
BTU Cottbus
Cottbus, Germany
denny.schneeweiss@tu-cottbus.de

Abstract

Our feature configuration tool S^2T^2 Configurator integrates (1) a visual interactive representation of the feature model and (2) a formal reasoning engine that calculates consequences of the user's actions and provides formal explanations. The tool's software architecture is designed as a chain of components, which provide mappings between visual elements and their corresponding formal representations. Using these mappings, consequences and explanations calculated by the reasoning engine are communicated in the interactive representation.

1. Introduction

In the research on feature models different aspects have been addressed. First, there is work on *formal semantics* of feature models [6], which enables us to precisely express the available configurations of a product line in the form of a feature model. Second, there is the *interactive configuration* of feature models, as addressed by visualization of feature models [2] or feature modeling tools [1]. In this paper we strive to link these two worlds. So how can we provide a usable feature model representation, which can be configured interactively and precisely implements the underlying formal semantics?

We address this problem with S^2T^2 Configurator, a research prototype which integrates an interactive visual representation of feature models and a formal reasoning engine.[1] The architecture of the Configurator is designed as a chain of components, which provide mappings between visual elements and their corresponding representations in the formal reasoning engine, see Figure 1. The *Software Engineer* interacts with multiple *Views* of the *Model*. The *Consequencer* infers consequences and provides explanations. A *Translator* serves as a mediator between the representations used in these components.

*This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre.

[1] S^2T^2 stands for " SPL of SPL Techniques and Tools ".

2. Requirements

Before we present the Configurator tool, we have to briefly discuss the required functionality. First, the application has to *load the model* and *translate it into a formal representation*. Subsequently, the user configures the model by *making and retracting decisions*. For Boolean feature models, a user decision is either a *selection* or an *elimination* of a certain feature. Hence, we have four potential configuration states (the power set of $\{true, false\}$): *Undecided*, *Selected*, *Eliminated*, and *Unsatisfiable*.

After any change (loading, user interaction) the tool has to *infer consequences*, taking into account constraints imposed by the model and user decisions. These consequences have to be communicated in the visual representation. We distinguish four sources of configuration: *M = Model* (given in the model), *MC = ModelConsequence* (consequence of M), *U = User* (given by interaction), and *UC = UserConsequence* (consequences of *U*, might rely on *M*).

The tool must *enforce constraints* and disable decisions that lead to configuration states where no valid configuration is possible without retracting decisions ("*backtrack freeness*"). The tool shall *explain* why certain configuration decisions were made automatically. The explanation shall be given within the model by highlighting elements that led to the explained decision.

3. Feature Model

The goal of interactive configuration of feature models led us to a particular design of our modeling language. To be able to map consequences and explanations generated by the Consequencer to visual representations we "chopped up" our feature models in smaller pieces which we call *feature model primitives*. For instance, to describe feature groups, we use primitives like `AlternativeGroup`,

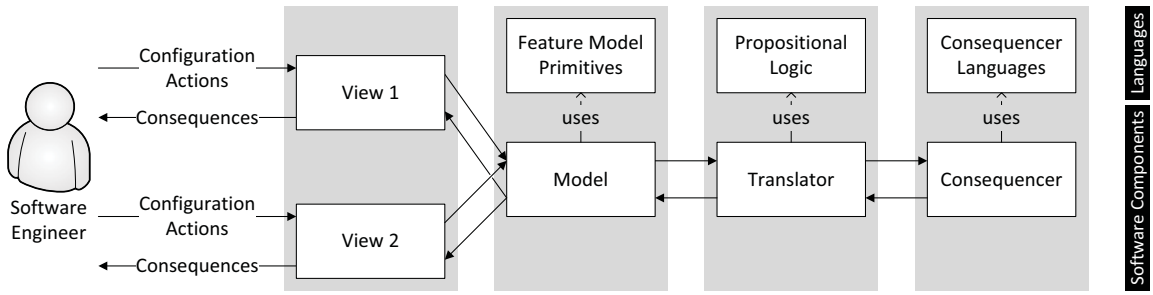


Figure 1. Overview of the components and languages used in S^2T^2 Configurator

`GroupHasParent`, or `GroupHasChild`. Similar primitives exists for other elements typically found in feature models (e.g., root, mandatory and optional subfeatures) and to capture user decisions (selection, elimination). Overall the `FeatureModel` consists of a set of `Features` and a set of `FeaturePrimitives`.

Features can be interpreted as variables and primitives as constraints over these variables. A legal configuration has to fulfill all of these constraints. Hence, if we interpret each primitive by translating it into a formal representation and conjoin all of these translations, this gives us the formal semantics of the overall model. We use a similar structure (variables + constraints) for other more formal languages. Consequently, (1) we can use a *generic design* for all the configurators operating upon these languages and (2) reasoning and explanations are implemented by a *chain of mappings* between constraints in various languages.

When the user starts configuring a model, his decisions can be described by adding primitives, e.g., `SelectedFeature` or `EliminatedFeature`. Since the tool only offers configuration decisions that keep the model in a consistent state, making decisions and adding the corresponding primitives will create a more constrained feature model, which represents a subset of feature configurations of the original model. During this process, the backtrack-freeness of the configurator guarantees that at least one legal configuration remains.

4. User Interface

The meta-model gives us the means for describing a feature model as a set of primitives. Let us see how this is presented to the user. Figure 2(a) shows an example feature model (based on [3]) in S^2T^2 Configurator right after loading.

The features `Car`, `KeylessEntry`, `Body`, `Gear` are mandatory and selected. The `Engine` configuration source for all of these primitives is *M = Model*. Because `Engine Requires Injection`, the configurator infers that the latter has to be selected as well and creates the corresponding `SelectedFeature`-primitive with configuration source $ModelConsequence(MC)$. The configuration states of features are represented as icons (check mark = selected, X = eliminated, empty box = undecided). Icons for features with the source M or MC are shown in gray to indicate that the user cannot change the state.

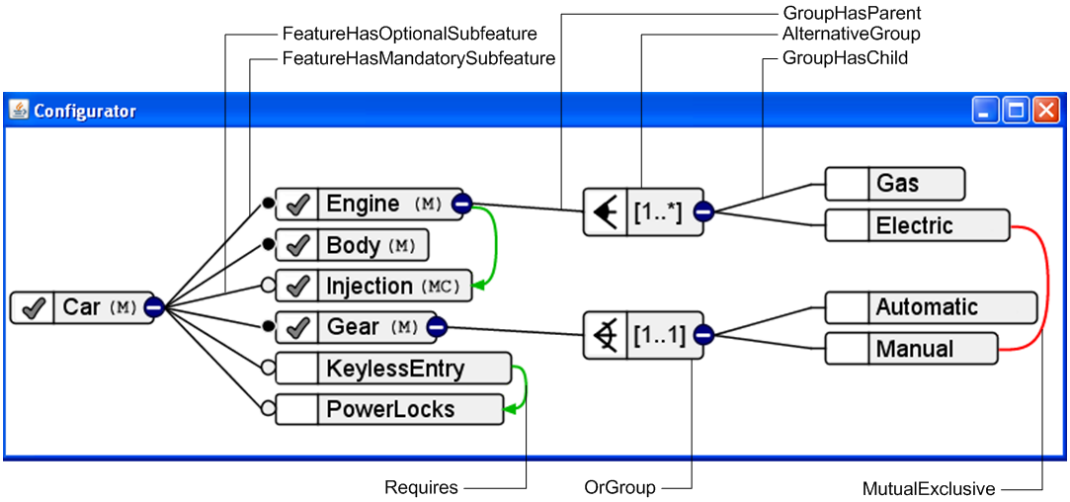
If the user now selects `KeylessEntry` the Consequencer deduces that `PowerLocks` has to be selected as well. Therefore, a `SelectedFeature(PowerLocks)` primitive is created and the view is updated accordingly (see Figure 2(b)).

The user might want to get an explanation why a certain feature was automatically selected or eliminated. This can be done via a pop up menu (see Figure 2(c)). When the user clicks `Explain`, the view queries the configurator for an explanation for the currently focussed feature. Thanks to our software design, the explanation can be mapped back to a list of primitives, which get highlighted in the view. For instance, when asking "Why is PowerLocks selected?" the tool will highlight `SelectedFeature(KeylessEntry)` and the corresponding `Requires({KeylessEntry}, {PowerLocks})`.

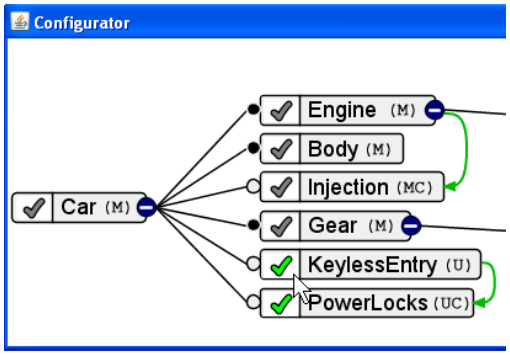
5. Integration between UI and Consequencer

One of our design goals was to allow *multiple* views, which can be used side-by-side, e.g., to focus on different aspects of the same model. Hence, when a configuration decision is made within one view, all resulting updates have to be propagated to the other views.

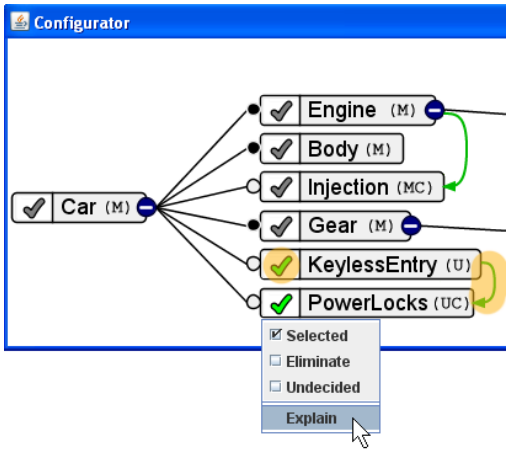
Hidden from the views the model communicates with the Consequencer. When a view commits a change to the model by adding or removing a primitive, this modification is first passed to the Consequencer, which produces consequences. These are then applied to the model. The modification (as triggered by the view) and the application of the consequences are performed atomically, in the sense that



(a) Visual representation of the feature model (call-outs indicate corresponding primitives from the meta-model).



(b) Configuration by interaction and consequences).



(c) Explanation of consequences.

Figure 2. Visual representation of the model in the view of S^2T^2 Configurator

no other operations are allowed before the consequences are applied. This is enforced by the interface, which all views must use to perform operations on the set of primitives. Thus the model is back in a valid state at the end of such an modification-consequence-combination. Subsequently all views are notified about the changes (including the inferred consequences).

6. Translator

The purpose of the translator is to get *from* feature model primitives *to* a format understood by the Consequencer, which is reasoning on some form of mathematical logic.

However, the Translator is not one-way. When providing consequences and explanations, it has to realize communication *from* the Consequencer *to* feature model primitives.

From a Software Engineering perspective, it is important that the tool can be easily used with *different reasoning engines* that realize the Consequencer component. Such engine typically has its own language as it can be used independently of the configurator.

To facilitate this, the Translator decomposes the translation process into several steps, each represented by a different component, a mini-configurator. Each of the mini-configurators communicates via certain language.

The following diagram depicts the mini-configurator chain as realized in the current implementation.

Feature Primitive Configurator (`FPC`) translates between feature primitives and propositional logic. Propositional

Logic Configurator (PLC) provides communication between propositional logic and Reasoning Engine Configurator (REC), which performs the actual reasoning.

The output of FPC is a machine-readable language of propositional logic with logic conjunctives and negation and thus amenable to further conversion to reasoning engines.

Each feature f corresponds to a Boolean variable V_f. And the translation of the primitives is done according to the traditional semantics of feature models [6]. The following table lists several examples.

primitive	logic formula
OptionalChild(c, p)	$V_c \rightarrow V_p$
MandatoryChild(c, p)	$V_c \leftrightarrow V_p$
SelectedFeature(f)	V_f
Excludes($\{a, b\}$)	$\neg(V_a \wedge V_b)$

Hence the input of the mini-configurator REC is propositional logic while its output is propositional logic in the Conjunctive Normal Form (CNF). This form is required by the reasoning engine used in the implementation (see Section 7). A different engine might require a different format and this mini-configurator would have to be replaced.

To obtain a uniform view on these languages, we assume that in each of them a sentence comprises a set of *constraints* and a set of *variables*. Depending on the particular language, we use different variables and constraints as shown in the following table.

language	variables	constraints
feature model	features	feature primitives
prop. logic	Boolean variables	prop. formulas

With respect using Boolean variables for formal representation, note that while in the current implementation the mappings between variables in the different languages are 1-to-1, in general, more complicated mappings may arise. For instance, if we model a variable with a larger domain by using multiple Boolean variables.

This uniform view on the used languages enables us to provide a generic interface which any of the mini-configurators (e.g., FPC) implements. This interface can be found in Figure 3. Constraints can be added and removed later using the methods addConstraint and removeConstraint.

```
interface IConfigurator<Variable, Constraint> {
  void addConstraint(/*@non_null*/Constraint c);
  void removeConstraint(/*@non_null*/Constraint c);
  Set<Constraint> computeConstraints(Variable v);
  Set<Constraint> explain(/*@non_null*/Constraint c);
}
```

Figure 3. Configurator interface

The method computeConstraints infers consequences that apply to the given variable while the method explain explains why a given consequence was inferred.

This architecture is rather flexible as any of the components can be easily replaced by another one as long as it implements the same interface and relies only on the instance of the IConfigurator interface of the succeeding component.

7. Consequencer

The reasoning engine used in our implementation relies on a SAT solver [5], which is why it requires the Conjunctive Normal Form. The engine has been developed in our previous work and more details can be found elsewhere [4]. To connect this reasoning engine to the component chain of S^2T^2 Configurator, it was merely necessary to provide the IConfigurator interface for it (see Figure 3).

A different reasoning engine (e.g., [7]), would be added analogously.

References

[1] D. Beuche. Variants and variability management with pure::variants. In *3rd Software Product Line Conference (SPLC 2004), Workshop on Software Variability Management for Product Derivation*, Boston, MA, August 2004.

[2] G. Botterweck, S. Thiel, D. Nestor, S. bin Abid, and C. Cawley. Visual tool support for configuring and understanding software product lines. In *12th International Software Product Line Conference (SPLC 2008)*, Limerick, Ireland, September 2008. ISBN 978-7695-3303-2.

[3] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference (SPLC 2007)*, pages 23–34, Washington, DC, USA, 2007. IEEE Computer Society.

[4] M. Janota. Do SAT solvers make good configurators? In *First Workshop on Analyses of Software Product Lines (ASPL '08)*, 2008. Available at http://www.isa.us.es/aspl08.

[5] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. CHAFF: Engineering an efficient SAT solver. In *39th Design Automation Conference (DAC '01)*, 2001.

[6] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *Requirements Engineering Conference, 2006. RE 2006. 14th IEEE International*, pages 136–145, 2006.

[7] S. Subbarayan. Integrating CSP decomposition techniques and BDDs for compiling configuration problems. In *Proceedings of the CP-AI-OR*. Springer-Verlag, 2005.

Using First Order Logic to Validate Feature Model

Abdelrahman O. Elfaki, Somnuk Phon-Amnuaisuk, Chin Kuan Ho
Center of Artificial Intelligent and Intelligent computing,
Multimedia University, Cyberjaya, Malaysia
abdelrahman.osman.06@mmu.edu.my,somnuk.amnuaisuk@mmu.edu.my,ckho@mmu.edu.my

Abstract

Feature Model (FM) is approved as a successful technique to model variability in Software Product Line (SPL), therefore it is very important to produce error-free FM. Inconsistency in FM is addressed as key challenge in validation of FM. This paper completes the knowledge-base(KB) method for validating FM by defining a new operation, namely inconsistency-prevention. First the inconsistency in FM is categorized into four groups as a prerequisite process for inconsistency-prevention. Then the scalability of KB method for validating FM is tested. Empirical results for each operation are presented and discussed. The empirical results are employed to illustrate scalability and applicability of the proposed knowledge-base method.

1. Introduction and Motivations

FM is considering as one of the successful methods for applying variability in SPL [1]. Therefore it is very important to produce error-free FM, this process is non-feasible manually. Inconsistency detection is introduced in [2] as a research challenge. Inconsistency occurs from contradictions in constraint dependency rules. It is very complicated because it has different formats and it can occur between groups of features or between individual features. In our previous work [3], we introduced a method to validate a FM based on mapping FM to first order logic, one-to-one mapping and represent domain engineering as a KB. The proposed method [3] defines and provides auto-support for three operations for validating FM, namely: explanation, dead feature detection, and inconsistency-detection. The proposed operation inconsistency-detection can detect only inconsistency between individual features. In this paper we enhance this operation by illustrating how to map all inconsistency formats to *one-to-one* format. And rather than detect inconsistency, we improve the proposed method by defining a new operation that aims at preventing inconsistency in FM, this process is explore the KB(domain engineering) and according to specific states new rules added to KB. The overall contribution of the proposed method is the validating of FM within domain engineering process.

Scalability is one of the main factors that define the applicability of the methods that deal with FM. Empirical results approved (in literature) to test scalability. In this paper we test the scalability of the proposed method (four operations).

Related work is discussed in section 2. In section 3 we define the new operation in FM validation (inconsistency-prevention). Empirical results are presented and discussed in section 4. Finally, discussion and conclusion in section 5.

2. Related Work

Mannion [4] was the first to connect propositional formulas to feature models., but the model did not detect the dead features or inconsistency. Zhang et al.[5] defined a meta-model of FM using UML core package. Zhang only validated consistency-check during configuration. Benavides et al. [6] proposed a method for mapping feature model into constraint solver. This model was mainly developed for the analysis of SPL-based FM, rather than validating it. Batory[7] proposed a coherent connection between FMs, grammar and propositional formulas, represented basic FMs using context–free grammars plus propositional logic. This connection allows arbitrary propositional constraints to be defined among features and enables off-the-shelf satisfible solvers to debug FM. Batory's method validated FM within application engineering process (in product derivation), and it detected one type of inconsistency (one-to-one) and

did not detect the dead features. Although Janota[8] used higher-order logic to reason feature models, unfortunately no real implementation has been described. Thang[9] defined a formal model in which each feature was separately encapsulated by a state transition model. The aim of the study is to improve consistency verification among features, there is no mention for inconsistency or dead features. Czarnecki[10] proposed a general template-based approach for mapping feature models, and used object-constraint language (OCL) in [11] to validated constraint rules. In [3] we proposed rules for consistency constraint check. These rules are different from other methods (to validate consistency check) by considering and dealing with variation point constraints.

Trinidad[12] defined a method to detect dead features based on finding all products and search for unused features. The idea is to automate error detection based on theory of diagnosis [13]. This model mapped FM to diagnose-model and used CSP to analyze FM. Our proposed method to detect dead features [3] has less cost than Trinidad's because it searches only in three predefined cases, i.e. in domain engineering process. Validating FM in domain engineering is one of the main contributions of our proposed method. Segura et al. [14] used atomic set as a solution for the simplification of FMs. Segura scaled the work using random data generated by FAMA [15]. White et al. [16] proposed a method for debugging feature model configurations. This method scales as models with over 5,000 features are randomly generated by FAMA. Our proposed method is a collection of predicates; therefore it has high degree of flexibility, e.g. it can be partitioned regarding specific features. As we proved in [3], the proposed method is the first that deals with inconsistency. Moreover it addresses the validation operations (inconsistency-detection, dead feature detection, and explanation. In this study, we define inconsistency-prevention as a fourth operation applied to domain engineering (rather than configure a solution and validate it), which enhances the maturity of SPL. In the next section we illustrated the fourth operation (of the proposed KB method [3]) inconsistency prevention.

3. Knowledge-base method to validate FM: Inconsistency-prevention Operation

In our previous work [3], we defined and illustrated three operations: i) explanation, ii) dead feature detection, and iii) inconsistency-detection. In this section inconsistency-prevention is defined as fourth operation, and later experiments (which are designed to evaluate scalability of our proposed operations) are explained and results are discussed. In addition to validating existing FMs, the proposed method can be used to prevent inconsistency in FM by adding new relations (exclude/require).

The following parts of this section are defining the prerequisite process and illustrating the rules of inconsistency-prevention operations.

3.1. Prerequisite Process

The prerequisite process for inconsistency-prevention and inconsistency-detection operations is converting all forms of inconsistency into *one-to-one* relation form.

Inconsistency Forms

Inconsistency in FM, could be categorized in four groups:

Many-to-Many inconsistency:

In many-to-many inconsistency a set requires other set while the required set excludes the first one. E.g.
($\{A_1,A_2,\ldots,A_n\}$ requires $\{B_1,B_2,\ldots,B_n\}$)and
($\{B_1,B_2,\ldots,B_n\}$ excludes$\{A_1,A_2,\ldots,A_n\}$)
Other possible scenario, a set can requires other set while some features of the required set excludes some features of the first one. e.g.:
((A,B,C) *require*s (D,E,F)) and ((G,F,H) *excludes* (A , B,C)).
The constraint dependency could be between two or more sets.

Many-to-One inconsistency:

A set of features has constraint dependency relation (require/exclude) with one feature while this feature has a contradiction relation to this set or to some of its elements. e.g.
((A,B,C) *require*s D) and (D *excludes* (B,C)).

One-to-Many inconsistency:

One feature has constraint dependency relation (require/exclude) with a set of features while this set has a contradiction relation to this feature.

One-to-One inconsistency:

One feature has a constraint dependency with one feature while the second feature has a contradiction relation to the first feature. e.g.(A *requires* B) and (B *excludes* A).

In [3], we defined and illustrated five rules to detect One-to-One inconsistency. To detect other forms of inconsistency we need first to extend Many-to-Many, Many-to-One, and One-to-Many to represented as feature-to-feature relation. The following rule extends forms of inconsistency to feature-to-feature relation:

$\forall i,j \ \{A_i \mid 1 <= i <= n\}$ ***relation*** $\{B_j \mid 1 <= j <= m\} \Rightarrow A_i$ ***relation*** B_j

Where *relation* represents constraint rule (excludes/requires).

Example:

Many-to-Many inconsistency ((A,B) *require* (D,E)) and ((G,E)*excludes*(A,B)) can be extended to ((A *requires* D) and (A *requires* E) and (B *requires* D*)* and(B require E) and (G *excludes* A) and (G *excludes* B) and (E *excludes* A) and (

E *excludes* B)) feature-to-feature relation.

3.2. Inconsistency-prevention Rules

Inconsistency in feature model is a relationship between features that cannot be true at the same time[2].

To avoid inconsistency (prevent inconsistency) the following rules are proposed:

i. $\forall x,y,z:type(x,variant) \wedge type(y,variant) \wedge requires\_v\_v(x,y) \wedge type(z,variant) \wedge requires\_v\_v(y,z) \Longrightarrow requires\_v\_v(x,z)$.

ii. $\forall x,y,z:type(x,variationpoint) \wedge type(y,variationpoint) \wedge requires\_vp\_vp(x,y) \wedge type(z,variationpoint) \wedge requires\_vp\_vp(y,z) \Longrightarrow requires\_vp\_vp(x,z)$.

iii. $\forall x,y,z:type(x,variant) \wedge type(y,variationpoint) \wedge requires\_v\_vp(x,y) \wedge type(z,variationpoint) \wedge requires\_vp\_vp(y,z) \Longrightarrow requires\_v\_vp(x,z)$.

iv. $\forall x,y,z:type(x,variant) \wedge type(y,variant) \wedge requires\_v\_v(x,y) \wedge type(z,variationpoint) \wedge requires\_v\_vp(y,z) \Longrightarrow requires\_v\_vp(x,z)$.

v. $\forall x,y,z:type(x,variant) \wedge type(y,variant) \wedge requires\_v\_v(x,y) \wedge type(z,variant) \wedge excludes\_v\_v(y,z) \Longrightarrow excludes\_v\_v(x,z)$.

vi. $\forall x,y,z:type(x,variationpoint) \wedge type(y,variationpoint) \wedge requires\_vp\_vp(x,y) \wedge type(z,variationpoint) \wedge excludes\_vp\_vp(y,z) \Longrightarrow excludes\_vp\_vp(x,z)$.

vii. $\forall x,y,z:type(x,variant) \wedge type(y,variant) \wedge requires\_v\_v(x,y) \wedge type(z,variationpoint) \wedge excludes\_v\_vp(y,z) \Longrightarrow excludes\_v\_vp(x,z)$.

The outputs of this operation are new constraint dependency rules added to the KB (domain engineering) to sustain the consistency.

4. The Experiment

We developed an algorithm to generate random FM. (predicates form).We have three assumptions: i) each variation point and variant has unique name, ii) each variation point is orthogonal, and iii) all variation points have the same number of variants. The main parameters are the number of variants and the number of variation points. The remaining eight parameters *(common variants*, *common variation points*, *variant requires variant*, *variant excludes variant*, *variation point requires variation point*, *variation point excludes variation points, variant requires variation point,* and *variant excludes variation point)* are defined as a percentage. The number of variant-related parameters (such as; common variant) is defined as a percentage of the number of variants. The number of variation point-related parameters (such as; variant requires variation point) is defined as a percentage of the number of variation points. For each number of variant/variation point we made ten experiments, and calculated execution time as average. The experiments were done with the range (1000-20000) variants, and percentage range of 10%, 25%, and 50%.

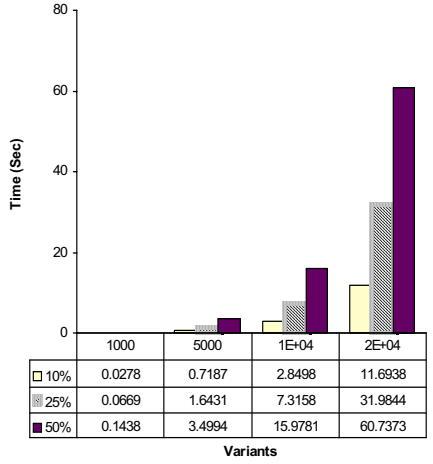
4.1. Empirical Results

4.1.1. Explanation: To evaluate the scalability of this operation, we define additional parameter, the predicate *select(V)*: where V is random variant. This predicate simulates user selection. Number of *select* predicate (defined as a percentage of number of variants) is added to the KB (domain engineering) for each experiment, and the variant *V* is defined randomly (within scope of variants). Figure 2 illustrates the average execution time.

4.1.2. Dead Feature Detection: Figure 3 illustrates the average execution time. For (*20,000)* variants and 50% of constraint dependency rules, the execution time is *3.423* minutes which can be considered good time, White et al.[16] scaled their work by 5,000 feature in one minute. The output of each experiment is a result file containing the dead variants.

4.1.3. Inconsistency-Detection: Figure 4 illustrates the average execution time to detect inconsistency in FM range from 1000 to 20,000 variants.

4.1.4 Inconsistency-prevention: new dependency rules (*requires*/*excludes*) should be added to the KB to prevent inconsistency. Figure 5 illustrates the average execution time to prevent inconsistency in FM range from 1000to20,000variants.



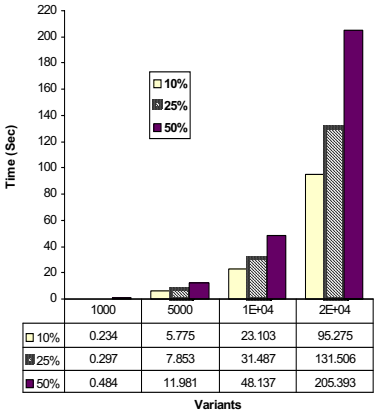
	1000	5000	1E+04	2E+04
10%	0.0278	0.7187	2.8498	11.6938
25%	0.0669	1.6431	7.3158	31.9844
50%	0.1438	3.4994	15.9781	60.7373

Figure 2: Explanation Results

5. Conclusion

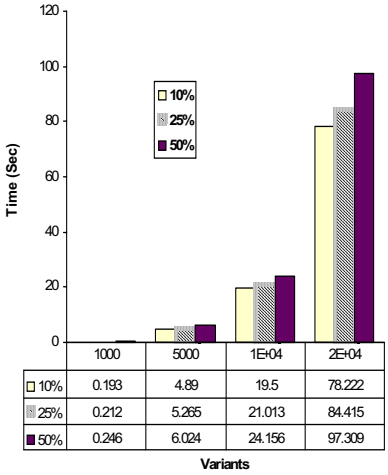
The proposed method deals with the complexity of validating product line based feature model. It is the first method that detects an inconsistency in FM of all types. Moreover, it explores the existing relations and prevents future inconsistency. The validation process (dead feature detection, inconsistency-detection, and inconsistency-prevention) should be applied to domain engineering which guarantees error free domain engineering. Error-free domain engineering (one of the main contributions of the proposed method) promises generation of valid applications. Many methods are applying empirical results to test scalability by generating random FMs [5, 13, 16]. Comparing with literature, our test range (1000 – 20,000 features) is

sufficient to test scalability.



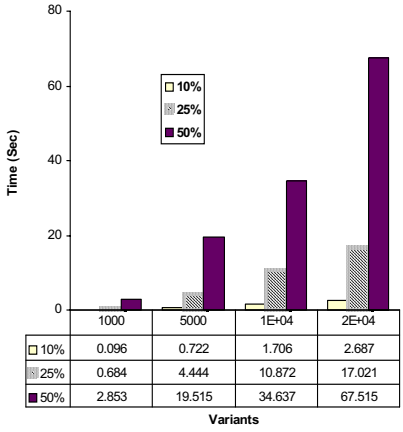
	1000	5000	1E+04	2E+04
10%	0.234	5.775	23.103	95.275
25%	0.297	7.853	31.487	131.506
50%	0.484	11.981	48.137	205.393

Figure 3: Dead-feature Detection Result



	1000	5000	1E+04	2E+04
10%	0.193	4.89	19.5	78.222
25%	0.212	5.265	21.013	84.415
50%	0.246	6.024	24.156	97.309

Figure4:Inconsistency-Detection



	1000	5000	1E+04	2E+04
10%	0.096	0.722	1.706	2.687
25%	0.684	4.444	10.872	17.021
50%	2.853	19.515	34.637	67.515

Figure 5: Inconsistency-prevention Results

6. References

[1] Krzysztof Czarnecki, "Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models", PhD Thesis Technical University of Ilmenau, October 1998.

[2] Don Batory, David Benavides, Antonio Ruiz-Cortés, *"Automated Analyses of Feature Models: Challenges Ahead"*, Special Issue on Software Product Lines ,Communications of the ACM, December 2006.

[3] Abdelrahman Osman. Elfaki , Somnuk Phon-Amnuaisuk, Chin Kuan Ho," Knowledge Based Method to Validate Feature Models", in the proceeding of 12th international conference of software product line, Limerick Ireland, 2008.

[4] M. Mannion , "Using First-Order Logic for Product Line Model Validation", Paper presented at the Second Software Product Line Conference (SPLC2), San Diego, CA. , 2002.

[5] Wei Zhang, H. Z., and Hong Mei, "A Propositional Logic-Based Method for Verification of Feature Models", Paper presented at the 6th International Conference on Formal Engineering Methods (ICFEM),2004.

[6] David Benavides, P. Trinidad, and A.Ruiz-Cortes, *"Automated Reasoning on Feature Models*", Advanced Information Systems Engineering (Vol. 3520/2005,), Springer, Berlin Heidelberg, 2005, pp. 491-503.

[7] Don Batory,*"Feature Models, Grammars, and Propositional Formulas",* Paper presented at the 9th International Software Product Lines Conference (SPLC05), Rennes, France, 2005.

[8] Mikolas Janota, Joseph Kiniry, *"Reasoning about Feature Models in Higher-Order Logic",* Paper presented at the 11th International Software Product Line Conference (SPLC07), 2007.

[9] Nguyen Truong Thang, "Incremental Verification of Consistency in Feature-Oriented Software", PhD thesis,Japan Advanced Institute of Science and Technology , September, 2005.

[10] Krzysztof Czarnecki, Michal Antkiewicz, *"Mapping features to models: A template approach based on superimposed variants",* Paper presented at the 4th International Conference on Generative Programming and Component Engineering (GPCE'05), Tallinn, Estonia, 2005.

[11] Krzysztof Czarnecki, Krzysztof Pietroszek, *"Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints"*, Paper presented at the 5th international conference on Generative programming and component engineering (GPCE'06), 2006.

[12] Pablo Trinidad, David Benavides, and Antonio Ruiz-Cort´es, *"Isolated features detection in feature models"*, Paper presented at the Advanced Information Systems Engineering (CAiSE), Luxembour, 2006.

[13] Pablo Trinidad , D. Benavides, A. Dura´n, A. Ruiz-Cortes,and M. Toro, *"Automated error analysis for the agilization of feature modeling"*, systems and software,doi:10.1016/j.jss.2007.10.030, 2008.

[14] Sergio Segura ," Automated Analysis of Feature Models using Atomic Sets", paper in proceeding of 12th international conference of software product line, Limerick Irland,2008.

[15] Pablo Trinidad, David Benavides, Antonio Ruiz-Cort?, Sergio Segura, Alberto Jimenez, "FAMA Framework," splc,pp.359, 2008 12th International Software Product Line Conference, 2008.

[16] Jules White, Doulas Schmidt, David Benvides, Pablo Trinidad, Antonio Ruiz-Cortes, "Automated Diagnosis of product line configuration errors on feature models", paper in proceeding of 12th international conference of software product line, Limerick Irland,2008.

VMWare: Tool Support for Automatic Verification of Structural and Semantic Correctness in Product Line Models

Camille Salinesi[1], Colette Rolland[1], Raúl Mazo[1,2]
[1] *CRI, Université Paris 1 – Sorbonne, 90, rue de Tolbiac, 75013 Paris, France*
[2] *Ingeniería & Software, Universidad de Antioquia, Medellín, Colombia*
{Camille.salinesi, Colette.Rolland}@univ-paris1.fr, raulmazo@gmail.com

Abstract

The verification of variability models is recognized as one of the key challenges for automated development of product lines. Some computational tools have been proposed to verify product line models and product line configurations models. VMWare is a tool integrating different criteria to verify structural and semantic correctness of models derived from the FORE metamodel. Our tool gives the possibility of (i) build feature-based product line models and product line configuration models, (ii) verify their structural and semantic correctness in a completely automated manner and (iii) import/export them in XMI files.

1. Introduction

Feature Modelling is a mechanism to represent requirements in the context of Software Product Lines (SPL). A Feature Model (FM) defines features and their usage constraints in product-lines (PL). Their main purposes are: (i) to capture feature commonalities and variabilities; (ii) to represent dependencies between features; and (iii) to determine combinations of features that are allowed and disallowed in the product line. A feature is a product characteristic that some stakeholders (e.g. users, sellers, engineers, customers) consider important to include in the description of the product line.

Automated analysis of FMs is recognized in the literature as an important challenge in PL engineering and is considered as an open issue by many SPL researchers [1], [2], [4], [10]. Verification of FMs is important for industry because any error in a Product Line Model (PLM) will inevitably affect the configuration models (PLCMs) and thereafter final products. By verification of FMs we mean the formal process of determining whether or not they satisfy well defined verification criteria. Verification criteria can be determined either by means of properties of the specification itself, or by means of a collection of properties of some other specification. FMs correctness includes structural correctness and semantic correctness.

This paper presents a prototype tool for PLMs and PLCMs construction and verification. The tool is based on a framework for the automated analysis of feature models. Broadly speaking, it allows: (i) creating PLMs and PLCMs; (ii) verifying structural correctness criteria of PLMs and PLCMs; (iii) verifying semantic correctness of PLMs; and (iv) verifying PLCMs in regard to PLMs. The implementation is based on a three-layer architecture and uses XMI files as a mechanism to exchange the FMs with other tools.

The remainder of the paper is structured as follows. Section 2 gives a brief overview of feature modeling and of the verification process. Section 3 describes the functionality and provides some implementations details of the framework. Section 4 concludes the paper and describes future works.

2. Feature Modeling and Verification

Feature modeling is the activity of identifying externally visible characteristics of products in a domain and organizing them into a feature model. The notation considered in this paper is FORE notation (Feature Oriented Requirements Engineering) [3].

The characteristics of the FORE notation are:

- a feature diagram is a Directed Acyclic Graph (DAG);
- a feature is represented by a node of this graph;
- relationships between features are represented by links. There are two types of relationship, namely variant dependency and transverse dependency;
- *variant dependencies* can be mandatory or optional. The collection of features related by variant dependencies take the form of a tree;

- *transverse dependencies* can be of two kinds: the *excluding* one or the *requiring* one;
- *optional* relationships with the same father can be grouped into a bundle. A relation can be member of one and only one bundle;
- a bundle has a cardinality that indicates the minimal and maximal number of features that can be chosen. The meaningful cardinalities are: 0..1, 1, 0..*N*, 1..*N*, *N*, *p*, 0..*p*, 1..*p*, *p*..*N*, *m*..*p*, 0..* and 1..*;
- graphically, a bundle of variant dependencies is represented by an arch that related all the implicated relations;

The FORE notation fits the construction of PLMs, while eliminating many ambiguities. However, there are no well established guidelines to identify structural and semantic errors in FORE models.

The FM verification process that we propose can be summarized in *Figure 1*. The process is structured around two cycles, the first one corresponds to PLMs verification and the second one corresponds to PLCMs verification.

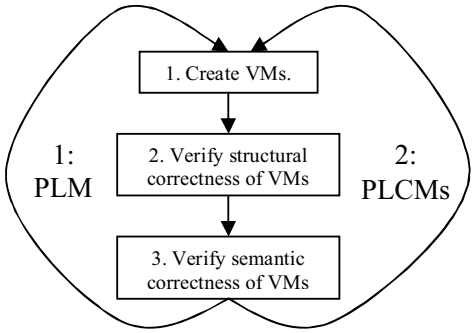


Figure 1. FORE-based PLMs and PLCMs correctness verification process.

2.1. Verify the structural correctness criteria of the Feature Model

Structural correctness concerns: (i) the correspondence between the model and the language in which the model is written; and (ii) the alignment between the model and a set of structural properties that any model of the same type must respect.

The purpose of the VMWare tool is to automatically verify FORE-based models according to a collection of well defined criteria [11]. To achieve this, we have divided the collection of criteria into three groups: (i) general criteria that every FORE-based FM shall respect; (ii) criteria specific to PLMs; and (iii) criteria specific to PLCMs.

In order to build a complete and consistent list of criteria, we undertake a state of the art of computational tools for construction of variability models supporting their automated verification. A summary of the criteria supported by the analysed tools are presented in Table 1.

Table 1. Structural and semantic correctness criteria (not) implemented in related tools.

	Tool	**VMWare**	**Feature Plugin** [5] / **DECIMAL** [8]	**XFeature**[1] [13] / **Pure::variants**[2] [7]	**Requiline**[3] [12]	**FAMA** [9]
Modeling Formalism		FORE Cons-traints	FODA* / Class	FORE	FORM	FORE
PLCM Verification		Y	Y	Y	Y	Y
PLM Verification		Y	N	Y	Y	Y
Criteria						
Structural	Root uniqueness	Y	N	Y	Y	N
	Child-father uniqueness	Y	Y	Y	Y	N
	Ordered cardinality	Y	N	N	N	N
	Applicable cardinality	Y	N	N	N	N
	Optional features and include dependencies coherence	Y	N	N	Y	Y
	Mandatory features and exclude dependencies coherence	Y	N	N	Y	Y
	Well limited cardinalities	Y	N	Y	N	N
	Consistency between transversal dependencies and cardinalities	Y	N	N	N	N
	No dead features	Y	N	N	N	Y
	DAG Structure	Y	?	Y	Y	N
Semantic	Richness – No void feature models	N	?	N	?	Y
	PLCM's compliance to the corresponding PLM	Y	Y	Y	Y	N
	Traceability	P	?	P	Y	?
	Uniqueness	N	?	?	N	N
	Pertinence	N	?	?	Y	N
	Modifiability	N	?	?	Y	?

Legend: Y = Yes, N = No, P = Partially, ? = unavailable information

* FODA with cardinality-based feature modeling.

[1]http://www.pnp-software.com/XFeature/

[2]http://www.software-acumen.com/purevariants/feature-models

[3]http://www-lufgi3.informatik.rwth-aachen.de/TOOLS/requiline

The criteria that we have chose for VMWare are defined bellow.

General criteria

1 *Root uniqueness*: The PLM should have only one root element.
2 *Child-father uniqueness*: A child feature should have one and only one father.
3 *Tree structure*: Variability structure of PLM, as well as PLCMs should be represented as connected and acyclic graphs.

PLM criteria

1 *Ordered cardinality*: All features grouped by a cardinality should be ordered in a consecutive manner.
2 *Applicable cardinality*: All features intervening in a cardinality should be optional.
3 *Optional features and include dependencies coherence*: This state of structural correctness criteria is respected when a feature is not at same time: mandatory and exclude dependent.
4 *Mandatory features and exclude dependencies coherence*: The state of structural correctness criteria is respected when a feature is not simultaneously: optional and require dependent.
5 *Well limited cardinality*: The state of structural correctness is respected when: (i) superior limit >= ||bundle||; and (ii) there are no cardinalities where both boundaries have 0 value (e.g. "0,0"), or the superior limit is lower than the inferior one, or where the inferior limit is a negative number.
6 *Consistency between transversal dependencies and cardinalities*: This criterion is determined by three conditions: (i) cardinality of bundle should be well formed; (ii) if a feature is involved in a bundle, then this feature cannot be related by a transverse relationship with other feature of the same bundle; and (iii) the same feature must not belong to two different bundles.
7 *No dead features*: It should be possible to include every feature in a PLM in at least one PLCM.
8 *DAG structure*: In a PLM it is forbidden to find a collection of features forming a cycle by means of *Transversal Dependencies* and/or *Variant Dependencies*. In order to evaluate this criterion, variability dependencies are enriched with a direction from the father to child. Transversal dependencies preserve its original directions. Thus, errors like exclusion (inclusion) of an ancestor and vice versa are identified.

Each of these criteria has been formally specified using first order logic predicates [11]. This allows implementing verification systematically using a SAT-like solver. For example, criterion ***child-father uniqueness*** was formally defined as follow:

$$\forall(feature\ P_i, feature\ C) \in PLM.\ FatherFeature(P_i) \wedge ChildFeature(C) \wedge (C\ childOf\ P_i) \wedge ((C\ Mandatory\ P_i) \oplus (C\ optional\ P_i)) \Rightarrow \left|(C \bullet P_i) \oplus (C \circ P_i)\right| = 1$$

Where "●" represents a mandatory and "○" an optional relationship between father and child features.

3. Implementation

The technologies used in the development process of our tool are:
(i) The Microsoft .NET Framework v2.0.50727 provided the general libraries.
(ii) Its source code was written using Microsoft Visual Studio 2005.
(iii) XmlExplorer Controls V1.0.0.0 was used in order to handle XML files, to record models and to handle interoperability with other CASE tools.

Functionalities

VMWare tool allows creating three types of specifications:
(i) Product line models using the FORE notation.
(ii) Product configuration models, in the adequate subset of FORE as described earlier.
(iii) Textual product line constraint specifications.

Our goal is to support the specification of other kinds of models such as goal models, aspect models, etc. A project is a set of several models, one by default. It includes the following functionalities:
1. Create a PLM.
2. Export and import PLM and PLCMs using an XMI file. This functionality allows communicating models from and to other applications.
3. Verify structural and semantic (partially) correctness of product line models.
4. Create and verify PLCMs, compared to a PLM. The set of verified criteria on PLCMs are: root uniqueness, child-father uniqueness, feature existence and PLM's constraint satisfaction.

Example

In VMWare, users can create or open either a project or a specific model. The "verification" menu offers to users the functions that allow choosing the different verification criteria. *Figure 2* gives an example of the feedback provided by the tool after the verification of the structural correctness of a FM. In *Figure 2*, *Feature*

1 and *Feature 2* are mandatory features that are linked by an *excludes*-type relationship.

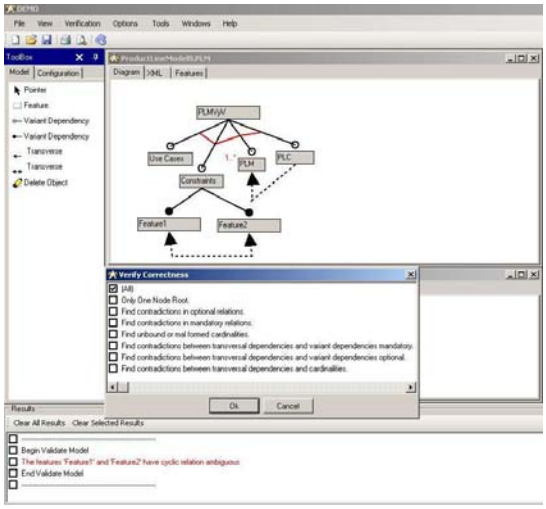


Figure 2. Identification of structural error in a PLM.

In order to verify semantic correctness of a PLM, it is necessary to check: (i) PLCMs' compliance to the corresponding PLM; and (ii) PLM's richness and traceability, uniqueness, pertinence, modifiability and usability of each feature. In order to check PLCMs' compliance, it is necessary to verify the *Feature existence* (every feature in a PLCM must also be a member of the PLM) and the *PLM's Constraint satisfaction* (PLCMs' structure must to be according to PLM's structure and restrictions). At this moment, we are working in formal definition of these criteria; they are not implemented in our tool yet.

4. Conclusions and Future Works

Our goal is to develop a generic method that would automatically help verify any kind of specification based on one or several VMs. We believe that the semantic verification criteria can be defined in a generic level at which any model can be checked. We are currently experimenting the use of constraint languages [6] on top of which these generic semantic verification criteria would be specified. The semantic verification process shall consist in a transformation of the verified model into a constraints program, and in a semantic verification of the constraints program. So far structural verification is concerned, we hope to be able to instantiate meta model-specific verification criteria from an ontology on generic criteria associated to an ontology on general meta-model concepts.

VMWare is not a mature tool yet, and many improvements remain, such us: (i) to support the definition and verification of VMs; (ii) to implant the multi-model verification criteria to validate consistency between PLM and PLCMs as well as between multiple PLMs; (iii) to implant other semantic correctness properties to verify and validate, like traceability, uniqueness, pertinence and modifiability of features and its relationships; and (iv) to support incremental verification.

References

[1] D. Batory, "Feature models, grammars, and propositional formulas", Software Product Lines Conference, LNCS 3714, pages 7–20, 2005.

[2] D. Batory, D. Benavides, and A. Ruiz-Cortés, "Automated analysis of feature models: Challenges ahead", Communications of the ACM, December, 2006.

[3] D. Streitferdt, "Family-Oriented Requirements Engineering", PhD Thesis, Technical University Ilmenau, 2003.

[4] Kim Lauenroth, Klaus Pohl, "Towards Automated Consistency Checks of Product Line Requirements Specifications", ACM/IEEE Intl. Conference on Automated Software Engineering, 2007, pp. 373-376.

[5] M. Antkiewicz, K. Czarnecki, "FeaturePlugin: feature modeling plug-in for Eclipse", OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, pp. 67-72.

[6] O. Djebbi, and C. Salinesi, "Towards an Automatic PL Requirements Configuration through Constraints Reasoning", Int. Workshop on Variability Modelling of Software-intensive Systems (VaMoS), Essen, Germany, January 2008.

[7] O. Spinczyk, D. Beuche, "Modeling and Building Software Product Lines with Eclipse", International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2004.

[8] P. Padmanabhan, R. Lutz, "Tool-Supported Verification of Product Line Requirements", Automated Software Engineering, Vol. 12, No. 4, 2005, pp. 447-465.

[9] P. Trinidad, A. Ruiz-Cortés, D. Benavides, S. Segura, A. Jimenez, "FAMA Framework", 12th Int. Software Product Line Conference (SPLC), 2008.

[10] Klaus Pohl, Gunter Bockle, Frank van der Linden, "Software Product Line Engineering: Foundations, Principles and Techniques", Springer, July 2005.

[11] Raul Mazo, Camille Salinesi, "Methods, techniques and tools for product line model verification. Research report", Centre de Recherche en Informatique CRI, Université Paris 1 Panthéon Sorbonne, 2008. In: http://halshs.archives-ouvertes.fr/docs/00/32/36/75/PDF/Methods_Techniques_and_Tools_for_PLM_Verification.pdf

[12] T. von der Maßen, H. Lichter, "RequiLine - A Requirements Engineering Tool for Software Product Lines", Software Product-Family Engineering, Springer LNCS 3014, 2004.

[13] V. Cechticky, A.Pasetti, O. Rohlik, and W. Schaufelberger, "Xml-based feature modelling", LNCS, Software Reuse: Methods, Techniques and Tools: 8th ICSR 2004. Proceedings, 3107:101–114, 2004.

A Tool for Modelling Variability at Goal Level

Christophe Gnaho[1, 2], Farida Semmak[1], Regine Laleau[1]
University of Paris XII - Val de Marne, LACL, France
[1] {semmak, laleau}@univ-paris12.fr
University of Paris Descartes, France
[2] gnaho@math-info.univ-paris5.fr

Abstract

This paper is a contribution to the improvement of requirements engineering in the context of the Cycab domain. The Cycab is a public vehicle with fully automated driving capability. In a previous work we proposed some extensions to the KAOS goal-oriented metamodel in order to enable explicit representation of variability at the early stage of requirements engineering. In this paper, we are interested in a software tool that implements the extended KAOS metamodel. The tool provides a GUI, which helps the designer to model Requirements Family Model (RFM) and then derive, according to the stakeholders needs, Specific Requirements Models (SRM) from the family model.

1. Introduction

This work is done as part of the TACOS1 project whose aim is to define a component-based approach to specify trustworthy systems from the requirements phase to the specification phase, in the Cycab transportation domain [1].

We need a requirements engineering approach, which addresses the early stage of requirements engineering during which stakeholders intentions are explored and different alternative ways to satisfy these intentions are investigated. Goal approaches have proven usefulness for that purpose.

Furthermore, the development of Cycab vehicles prototype takes time, requires frequent testing and leads to constant evolution. Therefore, the embedded software that makes the cycab vehicle run is subject to ongoing changes. Thus, Cycab software development would be greatly productive if design effort could be capitalized for reuse. One of the ways to accomplish this is through variability modelling.

[1] The TACOS project (Ref. ANR-06-SETI-017) is partially supported by the French National Research Agency

In previous work [5], we have presented some extensions to the KAOS goal oriented metamodel, in order to enable explicit representation of variability at goal level. This paper presents a software prototype that implement the extended metamodel. The tool helps the designer to build Requirements Family Model (RFM) and then to derive according to the stakeholders needs, Specific Requirements Models (SRM) from the family model.

The paper is organized as follows: Section 2 presents an overview of our approach. The implementation of the extended KAOS metamodel is presented in Section 3. Section 4 discusses related work. Finally, Section 5 concludes with some remarks about the results and future work.

2 Background

2.1 Overview of the supported approach

In the proposed approach, we attempted to apply reuse-based techniques at goal level. These techniques are inspired by the field of software product lines engineering [3] and domain engineering [4]. Figure 1 summarizes this approach.

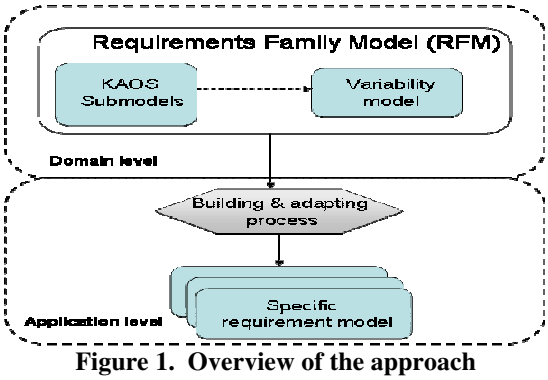


Figure 1. Overview of the approach

The domain level provides the *Requirements Family Model (RFM),* which enables the description of the large diversity of applications of the same domain by

identifying and expressing the common and variable requirements at goal level. The application level enables the building of specific requirements model. Its main component is the *Building and adapting process* that purpose is to derive the *specific requirement model* from the RFM, according to the needs of the stakeholders.

To specify a Requirements Family Model, we have chosen the KAOS (Knowledge Acquisition in autOmated Specification) goal-oriented approach [2]. However, this approach has not been originally designed to address variability-based systems. For this purpose, we proposed some extension in previous work in order to explicitly take into account variability concerns [5]. We call this extension Kaos Variability MetaModel (KVMM).

2.2 The KVMM

The main objective of the extensions made to KAOS is to make explicit 'what does vary' in the KAOS sub-models and 'how does it vary'. The 'what does vary' is captured thanks to the concept of *variation point* while the 'how does it vary' is described through the concepts of *facet and variant*. The concepts of facet and variant, together with the various relationships between them form the *variability model*. This model focuses on the relevant domain knowledge that presents multiple dimensions. It aims at structuring and organizing this knowledge for understanding and reusability.

The concept of **variation point** provides a means to make explicit variability in the KAOS sub-models. It relates these sub-models to the variability model

A facet is defined as a viewpoint or a dimension having an interest for a domain. A facet is described by the properties: name, description. For instance, the facet F1 has the name ": localization mode" and the description "to compute the vehicle position".

A facet is closely attached to one or more variants. **A variant** is defined as a way to realize a facet. For example, a Cycab may be localized by using a GPS sensor (Global Positioning System) or a WPS sensor (Wifi Positioning System) or an internal sensor or a combination of those three variants. Thus, to the facet "F1: localization mode" are attached the following variants: {V1: GPS, V2: WPS, V3: internal sensor}. A variant is described by the following properties: name, description, cost, rationale. For instance, one of the variant of the facet F1 has a name "GPS", a description "localization by measuring signal propagation time from different satellites", a cost "Ø" and a rationale "efficient if the localized area is not surrounded by high buildings".

3 Implementing the KAOS Variability MetaModel (KVMM)

In this section, we present a prototype implementation of the KVMM. The aim of this prototype is to provide guidance for the three following tasks: specification of a RFM as instance of the KVMM, generation of specific models from the RFM, and specification (and verification) of constrains over these models.

Three main options can be envisaged to develop this tool: (1) extending the KAOS Objectiver tool, (2) developing our own tool by using a generator for model editors such as Eclipse Modeling Framework (EMF) along with model transformation languages, (3) adapting a generic variability environment. We eliminated the first option because Objectiver is not an open source tool and we chose the third option because this solution significantly reduces the development effort. We adopted *XFeature* [6], [7] a generative environment for family instantiation to implement our prototype.

3.1 The modeling approach

XFeature provides a modeling approach that explicitly recognizes two main levels of modeling: family level and application level. Figure 2 shows an overview on how this modeling approach is applied to the KVMM.

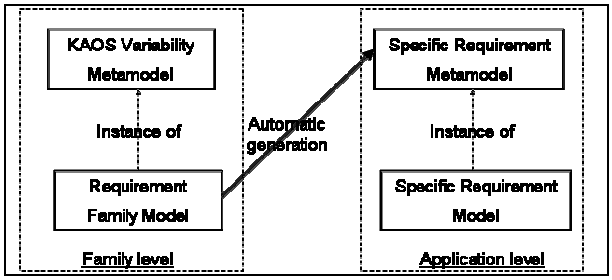


Figure 2 the modeling approach

At the family level, the user can create a Requirements Family Model (RFM) as instance of the KVMM. At the application level, the RFM is then automatically transformed thanks to transformation rules, in order to generate the Specific Requirement MetaModel (SRMM). This later defines the language that should be used to express the Specific Requirement Models (SRM).

In the present version, XFeature does not provide a tool to create and edit the KVMM, thus, it is therefore manually created and integrated.

3.2 The tool architecture

Each metamodel (KVMM and SRMM) is specified as an XML schema.

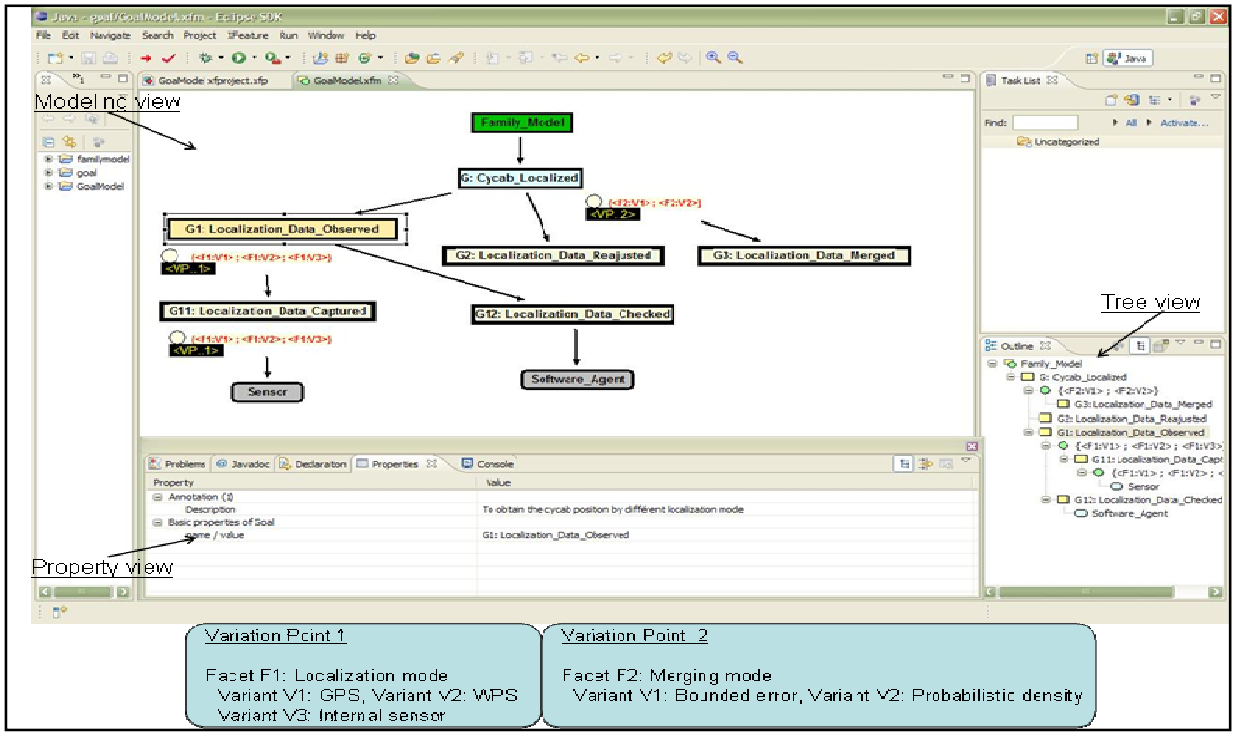


Figure 4: Model editors GUI

4 Related Work

It has been recognized that the effective variability modelling is dependent on the available tool support. In the context of SPL, many tools have been proposed to support variability modelling [6], [8], and [9]. Most of them have implemented models integrating feature trees, interrelations between features, cardinalities and so on. However, these models mainly focus on the specification phase. The prototype presented in this paper allows support of variability modeling at the early stage of requirements engineering.

5 Conclusion

In this paper, we have presented a software prototype to support variability modelling at goal level. This tool aims at helping the designer to create requirements model for a family of application and then derive specific requirements model. The implementation of this tool is based on Xfeature, a configurable environment that supports the creation of product family.

Future work should take place in two areas: (1) adapting Xfeature to automatically support the creation of KVMM, and (2) customizing the tool in order to obtain graphical notations similar to the ones proposed in Objectiver (Kaos tool).

6. References

[1] Parent, M., "Automated public vehicle: a first step towards the automatic highway", *Proc. Of the World Congress on Intelligent* , 1998

[2] Lamsweerde, A.: "From Systems Goals to Software Architecture", In Formal Methods for Software Architectures, *LNCS vol. 2804, Springer,* 2003

[3] Pohl, K., Bockle, G., van der Linden, F., *Software product line engineering: Foundations, Principles, and Techniques*, *Springer,* 2005

[4] Kang, S., Cohen, J., Hess, W., Novak, Peterson S.: "Feature-oriented domain analysis (FODA) feasibility study", CMU/SEI-90-TR-21, 1990

[5] Semmak, F. & Al, "Extended Kaos to support Variability for Goal oriented Requirements reuse", Int. Workshop Model Driven Information Systems Engineering with Caise'2008

[6] Cechticky V., & Al, 'XML-based Feature Modelling', *Software Reuse: Methods, Techniques, and Tools (ICSR)*, LNCS Series, Vol. 3107, Springer-Verlag, 2004

[7] Pasetti, A, & Al, technical note on a concept for the Xfeature tool, 2005

[8] Antkiewwiez, M & Czarnecki, K, "FeaturePlugin : Feature modeling Plug-In for Eclipse", OOPSLA'04, ETX Workshop, 2004

[9] Benavides, D, & Al, 'Tooling a framework for the Automated analysis of features models", 1st *Workshop on Vamos, Limerick, Ireland,* 2007

Previously published ICB - Research Reports

2008

No 28 (December 2008)
Goedicke, Michael; Striewe, Michael; Balz, Moritz: „Computer Aided Assessments and Programming Exercises with JACK“

No 27 (December 2008)
Schauer, Carola: “Größe und Ausrichtung der Disziplin Wirtschaftsinformatik an Universitäten im deutschsprachigen Raum - Aktueller Status und Entwicklung seit 1992”

No 26 (September 2008)
Milen, Tilev; Bruno Müller-Clostermann: “ CapSys: A Tool for Macroscopic Capacity Planning”

No 25 (August 2008)
Eicker, Stefan; Spies, Thorsten; Tschersich, Markus: “Einsatz von Multi-Touch beim Softwaredesign am Beispiel der CRC Card-Methode”

No 24 (August 2008)
Frank, Ulrich: “The MEMO Meta Modelling Language (MML) and Language Architecture – Revised Version”

No 23 (January 2008)
Sprenger, Jonas; Jung, Jürgen: “Enterprise Modelling in the Context of Manufacturing – Outline of an Approach Supporting Production Planning”

No 22 (January 2008)
Heymans, Patrick; Kang, Kyo-Chul; Metzger, Andreas, Pohl, Klaus (Eds.): “Second International Workshop on Variability Modelling of Software-intensive Systems”

2007

No 21 (September 2007)
Eicker, Stefan; Annett Nagel; Peter M. Schuler: “Flexibilität im Geschäftsprozess-management-Kreislauf”

No 20 (August 2007)
Blau, Holger; Eicker, Stefan; Spies, Thorsten: “Reifegradüberwachung von Software”

No 19 (June 2007)
Schauer, Carola: “Relevance and Success of IS Teaching and Research: An Analysis of the ‚Relevance Debate’

No 18 (May 2007)
Schauer, Carola: “Rekonstruktion der historischen Entwicklung der Wirtschaftsinformatik: Schritte der Institutionalisierung, Diskussion zum Status, Rahmenempfehlungen für die Lehre”

No 17 (May 2007)
Schauer, Carola; Schmeing, Tobias: “Development of IS Teaching in North-America: An Analysis of Model Curricula”

No 16 (May 2007)
Müller-Clostermann, Bruno; Tilev, Milen: "Using G/G/m-Models for Multi-Server and Mainframe Capacity Planning"

No 15 (April 2007)
Heise, David; Schauer, Carola; Strecker, Stefan: "Informationsquellen für IT-Professionals – Analyse und Bewertung der Fachpresse aus Sicht der Wirtschaftsinformatik"

No 14 (March 2007)
Eicker, Stefan; Hegmanns, Christian; Malich, Stefan: "Auswahl von Bewertungsmethoden für Softwarearchitekturen"

No 13 (February 2007)
Eicker, Stefan; Spies, Thorsten; Kahl, Christian: "Softwarevisualisierung im Kontext serviceorientierter Architekturen"

No 12 (February 2007)
Brenner, Freimut: "Cumulative Measures of Absorbing Joint Markov Chains and an Application to Markovian Process Algebras"

No 11 (February 2007)
Kirchner, Lutz: "Entwurf einer Modellierungssprache zur Unterstützung der Aufgaben des IT-Managements – Grundlagen, Anforderungen und Metamodell"

No 10 (February 2007)
Schauer, Carola; Strecker, Stefan: "Vergleichende Literaturstudie aktueller einführender Lehrbücher der Wirtschaftsinformatik: Bezugsrahmen und Auswertung"

No 9 (February 2007)
Strecker, Stefan; Kuckertz, Andreas; Pawlowski, Jan M.: "Überlegungen zur Qualifizierung des wissenschaftlichen Nachwuchses: Ein Diskussionsbeitrag zur (kumulativen) Habilitation"

No 8 (February 2007)
Frank, Ulrich; Strecker, Stefan; Koch, Stefan: "Open Model - Ein Vorschlag für ein Forschungsprogramm der Wirtschaftsinformatik (Langfassung)"

2006

No 7 (December 2006)
Frank, Ulrich: "Towards a Pluralistic Conception of Research Methods in Information Systems Research"

No 6 (April 2006)
Frank, Ulrich: "Evaluation von Forschung und Lehre an Universitäten – Ein Diskussionsbeitrag"

No 5 (April 2006)
Jung, Jürgen: "Supply Chains in the Context of Resource Modelling"

No 4 (February 2006)
Lange, Carola: "Development and status of the Information Systems / Wirtschaftsinformatik discipline: An interpretive evaluation of interviews with renowned researchers, Part III – Results Wirtschaftsinformatik Discipline"

2005

No 3 (December 2005)

Lange, Carola: "Development and status of the Information Systems / Wirtschaftsinformatik discipline: An interpretive evaluation of interviews with renowned researchers, Part II – Results Information Systems Discipline"

No 2 (December 2005)

Lange, Carola: "Development and status of the Information Systems / Wirtschaftsinformatik discipline: An interpretive evaluation of interviews with renowned researchers, Part I – Research Objectives and Method"

No 1 (August 2005)

Lange, Carola: „Ein Bezugsrahmen zur Beschreibung von Forschungsgegenständen und -methoden in Wirtschaftsinformatik und Information Systems"

Research Group	Core Research Topics
Prof. Dr. H. H. Adelsberger Information Systems for Production and Operations Management	E-Learning, Knowledge Management, Skill-Management, Simulation, Artificial Intelligence
Prof. Dr. P. Chamoni MIS and Management Science / Operations Research	Information Systems and Operations Research, Business Intelligence, Data Warehousing
Prof. Dr. F.-D. Dorloff Procurement, Logistics and Information Management	E-Business, E-Procurement, E-Government
Prof. Dr. K. Echtle Dependability of Computing Systems	Dependability of Computing Systems
Prof. Dr. S. Eicker Information Systems and Software Engineering	Process Models, Software-Architectures
Prof. Dr. U. Frank Information Systems and Enterprise Modelling	Enterprise Modelling, Enterprise Application Integration, IT Management, Knowledge Management
Prof. Dr. M. Goedicke Specification of Software Systems	Distributed Systems, Software Components, CSCW
Prof. Dr. R. Jung Information Systems and Enterprise Communication Systems	Process, Data and Integration Management, Customer Relationship Management
Prof. Dr. T. Kollmann E-Business and E-Entrepreneurship	E-Business and Information Management,E-Entrepreneurship/ E-Venture, Virtual Marketplaces and Mobile Commerce, Online-Marketing
Prof. Dr. B. Müller-Clostermann Systems Modelling	Performance Evaluation of Computer and Communication Systems, Modelling and Simulation
Prof. Dr. K. Pohl Software Systems Engineering	Requirements Engineering, Software Quality Assurance, Software-Architectures, Evaluation of COTS/Open Source-Components
Prof. Dr.-Ing. E. Rathgeb Computer Networking Technology	Computer Networking Technology
Prof. Dr. A. Schmidt Pervasive Computing	Pervasive Computing, Uniquitous Computing, Automotive User Interfaces, Novel Interaction Technologies, Context-Aware Computing
Prof. Dr. R. Unland Data Management Systems and Knowledge Representation	Data Management, Artificial Intelligence, Software Engineering, Internet Based Teaching
Prof. Dr. S. Zelewski Institute of Production and Industrial Information Mangement	Industrial Business Processes, Innovation Management, Information Management, Economic Analyses

For more information visit us on the Web: **http://www.icb.uni-due.de**

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